Cammini minimi

Gianluigi Zavattaro
Dip. di Informatica – Scienza e Ingegneria
Università di Bologna
gianluigi.zavattaro@unibo.it

Definizione del problema

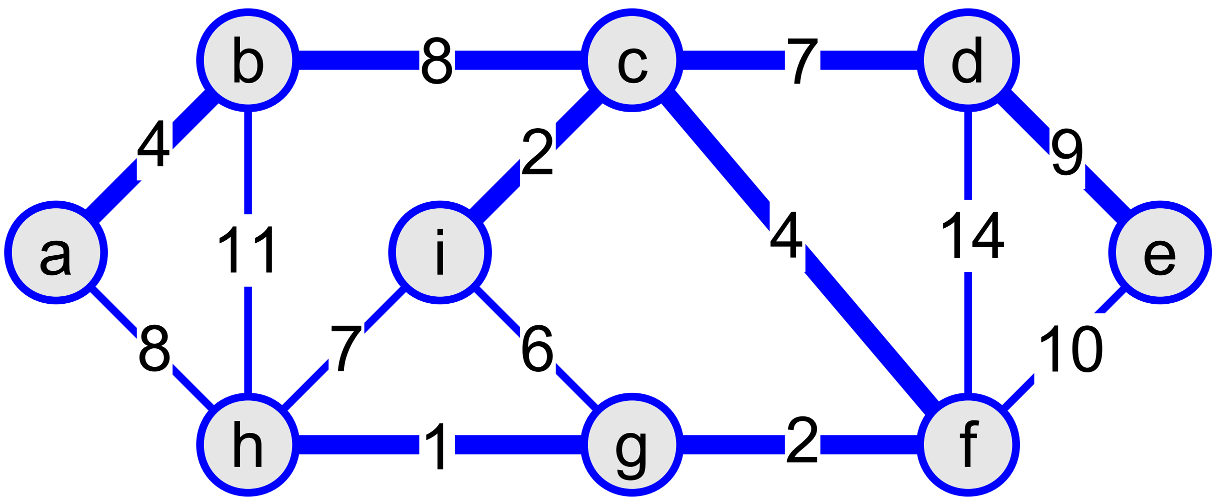
- Consideriamo un grafo orientato $G = (V, E)$ in cui ad ogni arco $(x, y) \in E$ sia associato un costo $w(x, y)$
- Il costo di un cammino $\pi = (v_0, v_1, \ldots v_k)$ che collega il nodo v_0 con v_k è definito come

$$w(\pi) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

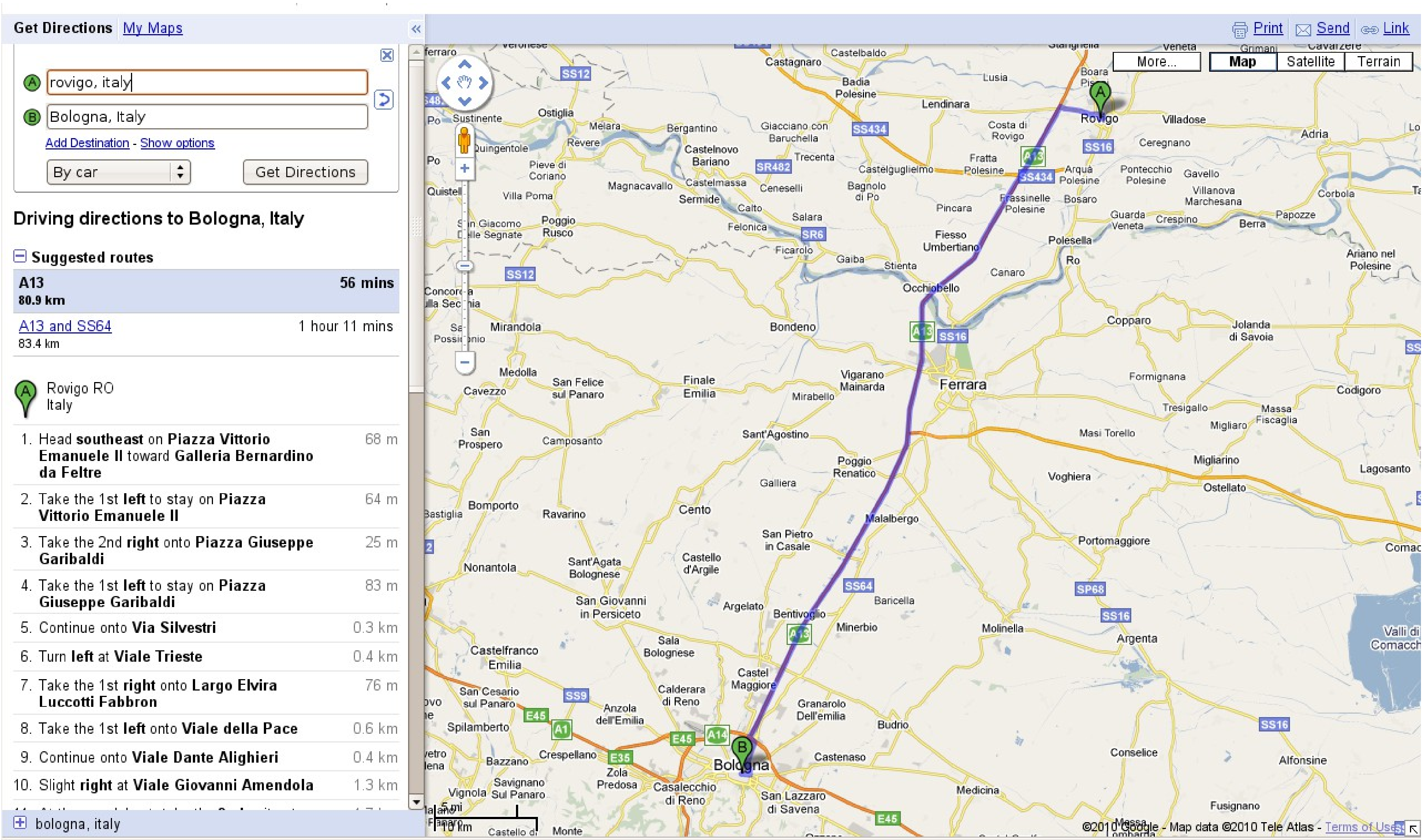
- Data una coppia di nodi v_0 e v_k, vogliamo trovare (se esiste) un cammino $\pi_{v0\ vk}^*$ di costo minimo tra tutti i cammini che vanno da v_0 a v_k

Osservazione

- Il problema del MST e dei cammini di costo minimo sono due problemi differenti
  - Es: il cammino di costo minimo che collega *h* e *i* è (*h*, *i*) oppure (*h*, *g*, *i*), entrambi di peso 7
  - In questo caso il cammino di costo minimo non fa parte del MST



Applicazioni



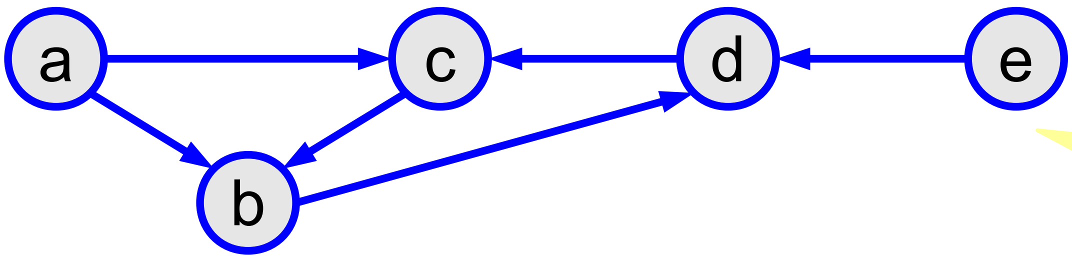
Diverse formulazioni del problema

1. Cammino di costo minimo fra una singola coppia di nodi *u* e *v*
   - Determinare, se esiste, un cammino di costo minimo π_{uv}^* da *u* verso *v*
2. Single-source shortest path
   - Determinare cammini di costo minimo da un nodo sorgente *s* a tutti i nodi raggiungibili da *s*
3. All-pairs shortest paths
   - Determinare cammini di costo minimo tra ogni coppia di nodi *u*, *v*

- Non è noto alcun algoritmo in grado di risolvere il problema (1) senza risolvere anche (2) nel caso peggiore

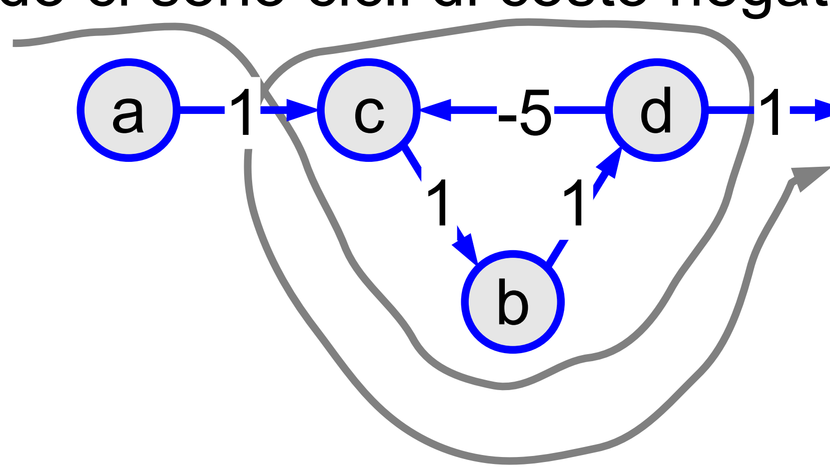
Osservazione

- In quali situazioni non esiste un cammino di costo minimo?
  - Quando la destinazione non è raggiungibile



*Non esiste alcun cammino che connette **a** con **e***

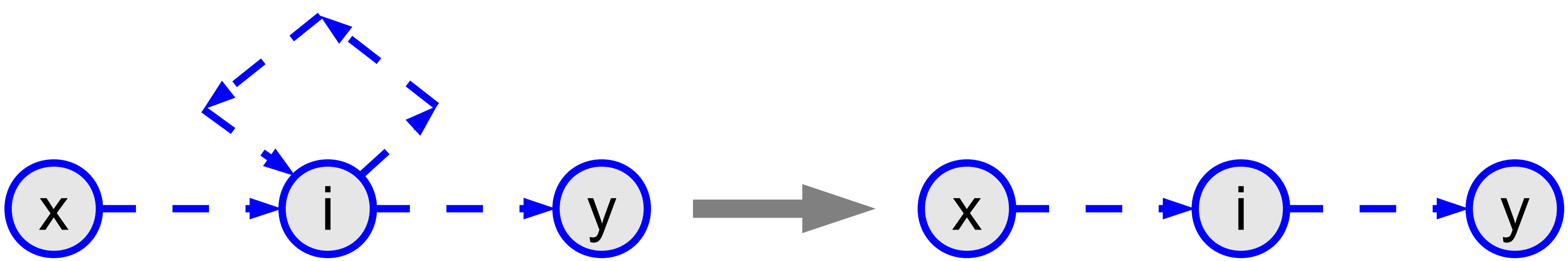
  - Quando ci sono cicli di costo negativo



*È sempre possibile trovare un cammino di costo inferiore che connette **a** con **e***

Esistenza

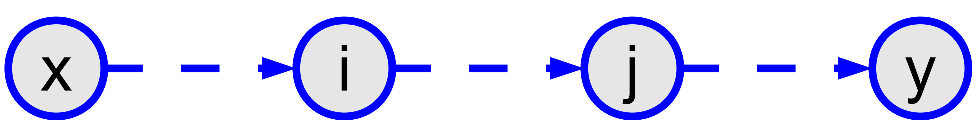
- Sia $G = (V, E)$ un grafo orientato con funzione peso w. Se non ci sono cicli negativi, allora fra ogni coppia di vertici connessi in G esiste sempre un cammino semplice di costo minimo
- Dimostrazione
  - Possiamo sempre trasformare un cammino in un cammino semplice (privo di cicli)



  - Ogni volta che si rimuove un ciclo, il costo diminuisce (o resta uguale), perché per ipotesi non ci sono cicli negativi
  - Il numero di cammini semplici e' finito, esiste un minimo

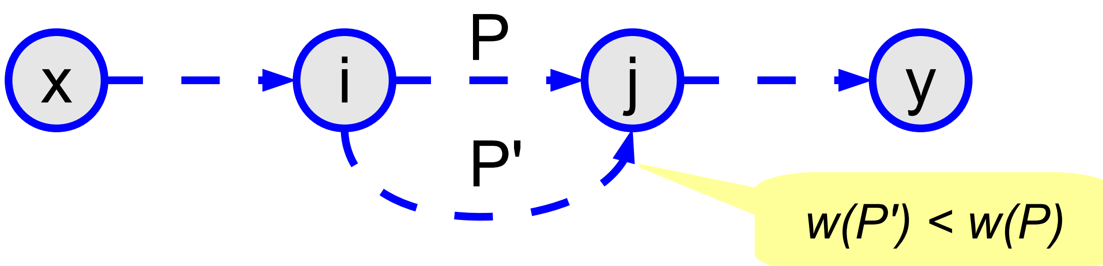
Proprietà (sottostruttura ottima)

- Sia $G = (V, E)$ un grafo orientato con funzione costo w; allora ogni sotto-cammino di un cammino di costo minimo in G è a sua volta un cammino di costo minimo
- Dimostrazione
  - Consideriamo un cammino minimo $\pi_{xy}{}^*$ da x a y
  - Siano i e j due nodi intermedi
  - Dimostriamo che il sotto-cammino di $\pi_{xy}{}^*$ che collega i e j è un cammino minimo tra i e j

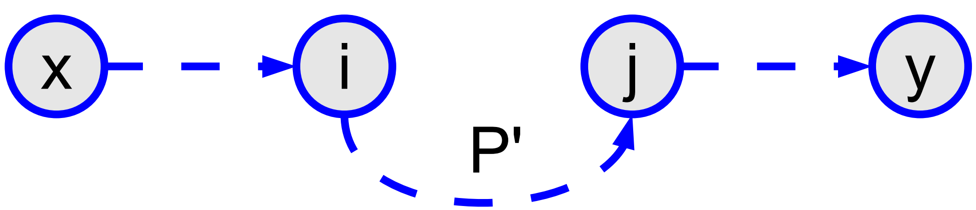


Proprietà (sottostruttura ottima)

- Supponiamo per assurdo che esista un cammino P' tra *i* e *j* di costo strettamente inferiore a P



- Ma allora potremmo costruire un cammino tra *x* e *y* di costo inferiore a π_{xy}*, il che è assurdo perché avevamo fatto l'ipotesi che π_{xy}* fosse il cammino di costo minimo



Albero dei cammini di costo minimo

- Sia *s* un vertice prefissato di un grafo orientato pesato $G = (V, E)$. Allora esiste un albero *T* che contiene i vertici raggiungibili da *s* tale che ogni cammino in *T* sia un cammino di costo minimo
  - Entrambi gli algoritmi che studieremo (Bellman-Ford e Dijkstra) restituiscono anche un albero come l'albero *T* descritto sopra
    - Quindi oltre ad indicare il costo complessivo minimo per raggiungere una certa destinazione, l'algoritmo indicherà anche una istanza di cammino con tale costo

Distanza tra vertici in un grafo

- Sia $G = (V, E)$ un grafo orientato con funzione costo w. La distanza d_{xy} tra x e y in G è il costo di un cammino di costo minimo che li connette; $+\infty$ se tale cammino non esiste

$$d_{xy} = \begin{cases} w(\pi^*_{xy}) & \text{se esiste un cammino di costo minimo } \pi^*_{xy} \\ +\infty & \text{altrimenti} \end{cases}$$

- **Nota**: $d_{vv} = 0$ per ogni vertice v
- **Nota**: Vale la disuguaglianza triangolare

$$d_{xz} \leq d_{xy} + d_{yz}$$

Condizione di Bellman

- Per ogni arco (*u*, *v*) e per ogni vertice *s*, vale la seguente disuguaglianza

$$d_{sv} \leq d_{su} + w(u, v)$$

- Dimostrazione
  - Dalla disuguaglianza triangolare si ha

$$d_{sv} \leq d_{su} + d_{uv}$$

  - Ma risulta anche

$$d_{uv} \leq w(u, v)$$

la distanza minima tra *u* e *v* non puo' essere maggiore del costo dell'arco (*u*, *v*) !

da cui la tesi

Trovare cammini di costo minimo

- Dalla condizione di Bellman

$$d_{sv} \leq d_{su} + w(u, v)$$

si può dedurre che l'arco (*u*, *v*) fa parte di un cammino di costo minimo π_{sv}* se e solo se

$$d_{sv} = d_{su} + w(u, v)$$

Tecnica del *rilassamento*

- Supponiamo di mantenere una stima $D_{sv} \geq d_{sv}$ della lunghezza del cammino di costo minimo tra *s* e *v*
- Effettuiamo dei passi di "rilassamento", riducendo progressivamente la stima finché si ha $D_{sv} = d_{sv}$

```
if (D_su + w(u,v) < D_sv) then D_sv ← D_su + w(u,v)
```

Algoritmo di Bellman e Ford

- Considieriamo un cammino di costo minimo inizialmente ignoto $\pi_{s\,vk}{}^* = (s, v_1, \ldots v_k)$
- Sappiamo che $d_{sv_k} = d_{sv_{k-1}} + w(v_{k-1}, v_k)$

  da cui partendo da $D_{ss} = 0$ (e $D_{st} = +\infty$, per $t \neq s$), *potremmo* effettuare i passi di rilassamento seguenti

$$
\begin{array}{lcl}
D_{sv_1} & \leftarrow & D_{ss} + w(s, v_1) \\
D_{sv_2} & \leftarrow & D_{sv_1} + w(v_1, v_2) \\
 & \vdots & \\
D_{sv_k} & \leftarrow & D_{sv_{k-1}} + w(v_{k-1}, v_k)
\end{array}
$$

Algoritmo di Bellman e Ford

- Problema: noi non conosciamo gli archi del cammino minimo $\pi_{s\,vk}*$ né il loro ordine, quindi non possiamo fare il rilassamento nell'ordine corretto
- Però se eseguiamo per ogni arco (u, v)

```
if (Dsu + w(u,v) < Dsv) then Dsv ← Dsu + w(u,v)
```

sicuramente includeremo anche il primo passo di rilassamento “corretto”

$$D_{sv_1} \leftarrow D_{ss} + w(s, v_1)$$

Algoritmo di Bellman e Ford

- Ad ogni passo consideriamo tutti gli *m* archi del grafo (*u*, *v*) ed effettuiamo il passo di rilassamento

```
if (D_su + w(u,v) < D_sv) then D_sv ← D_su + w(u,v)
```

- Dopo *n* - 1 iterazioni (tante quanti sono i possibili vertici di destinazione dei cammini che partono da *s*) siamo sicuri di aver calcolato tutti i valori $D_{s\,v_k}$ corretti

Algoritmo di Bellman e Ford

single-source shortest path

```
double[1..n] BellmanFord(Grafo G=(V,E,w), int s)
    int n ← G.numNodi();
    int pred[1..n], v, u;
    double D[1..n];
    for v ← 1 to n do
        D[v] ← +∞;
        pred[v] ← -1;
    endfor
    D[s] ← 0;
    for int i ← 1 to n - 1 do
        for each (u,v) in E do
            if ( D[u] + w(u,v) < D[v] ) then
                D[v] ← D[u] + w(u,v);
                pred[v] ← u;
            endif
        endfor
    endfor
    // eventuale controllo per cicli negativi (vedi seguito)
    return D;
```

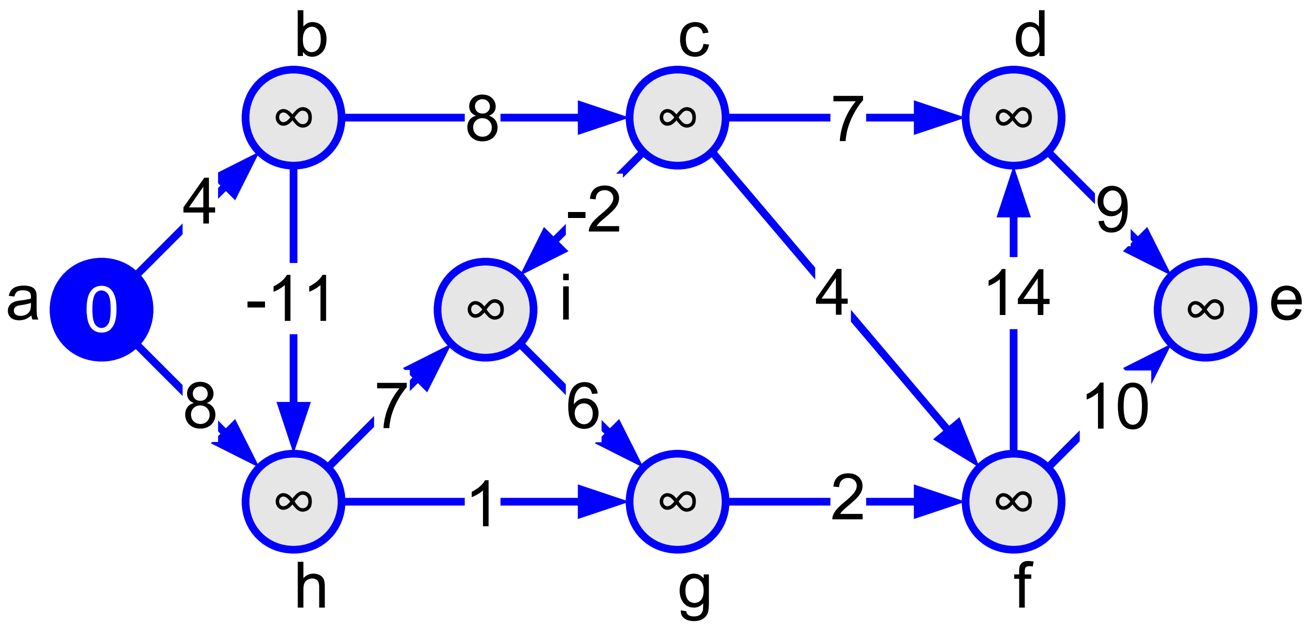
I nodi del grafo sono identificati dagli interi 1, … n

D[v] = (stima della) distanza del nodo v dalla sorgente s

pred[v] = predecessore del nodo v sul cammino di costo minimo che collega s con v

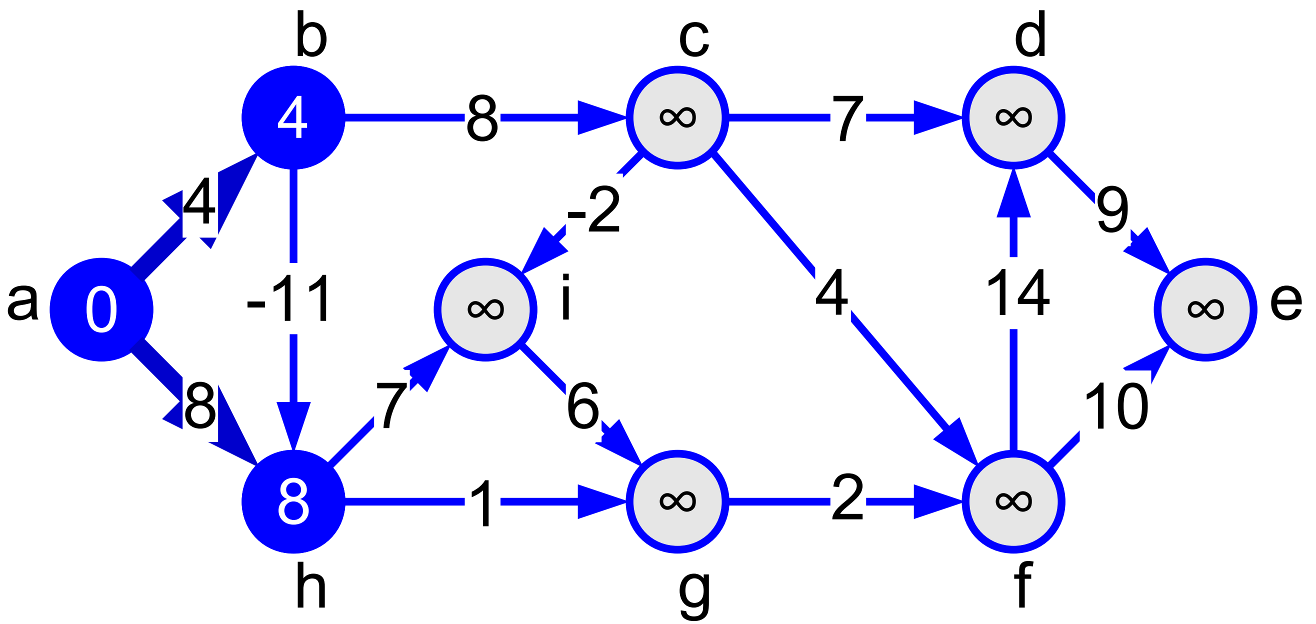
- Costo $O(nm)$

Esempio di esecuzione



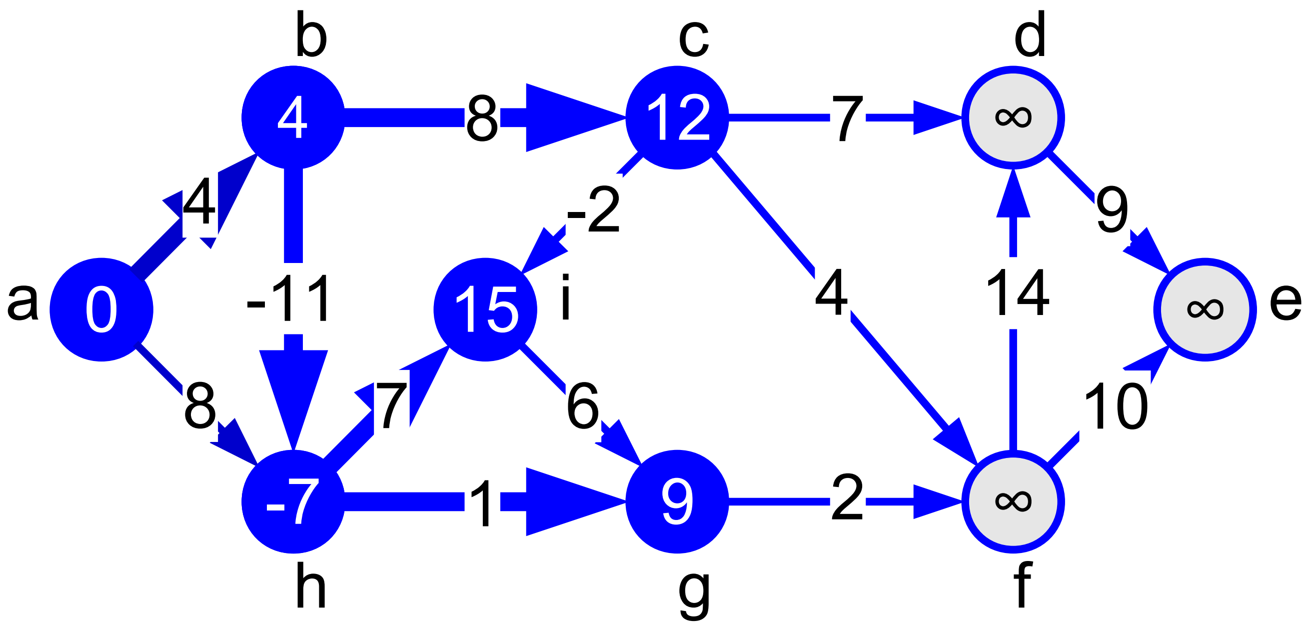
Esempio di esecuzione

Assumiamo che (a,b) e (a,h) siano gli ultimi archi considerati al ciclo 1



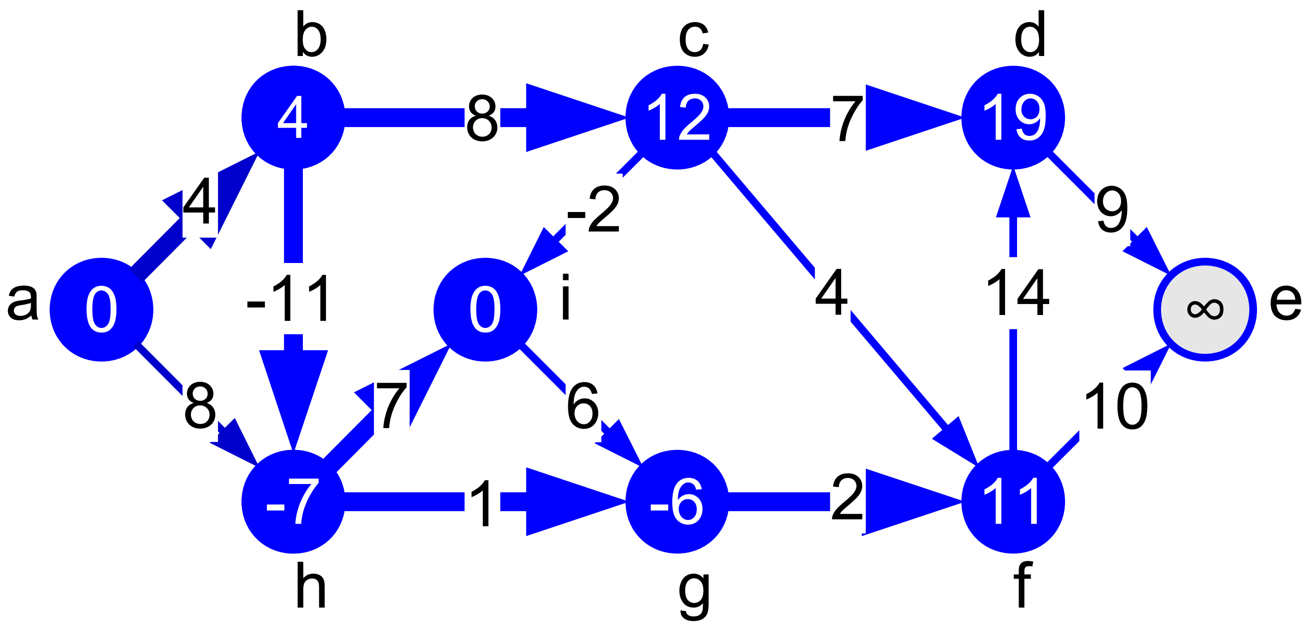
Esempio di esecuzione

Assumiamo che (b,c), (h,i), (h,g) ed infine (b,h) siano gli ultimi archi considerati al ciclo 2



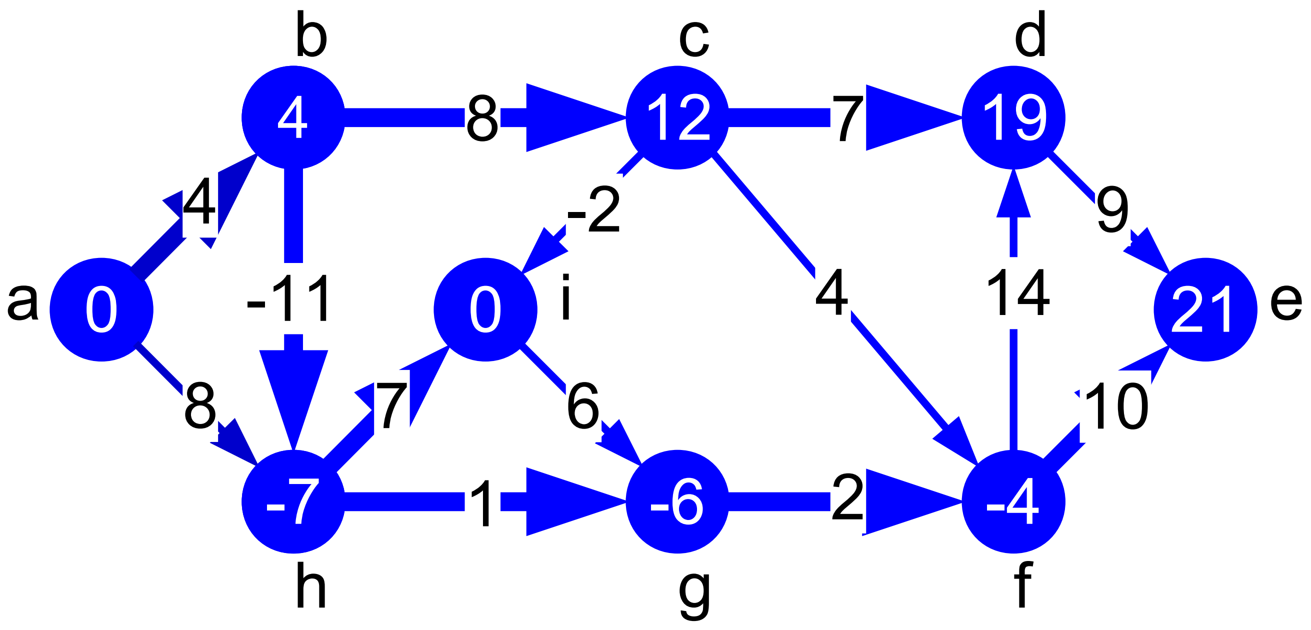
Esempio di esecuzione

Assumiamo che (c,d), (g,f) ed infine (h,g) siano gli ultimi archi considerati al ciclo 3

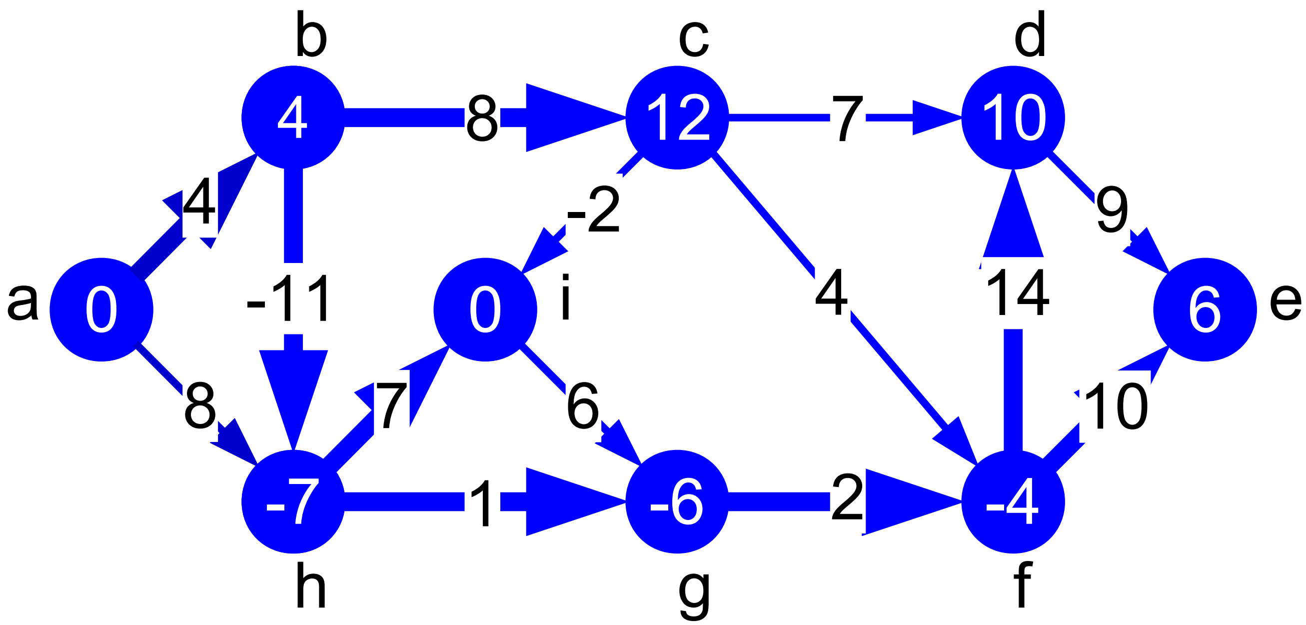


Esempio di esecuzione

Assumiamo che (g,f) sia l'ultimo arco considerati al ciclo 4



Esempio di esecuzione



Algoritmo di Bellman e Ford

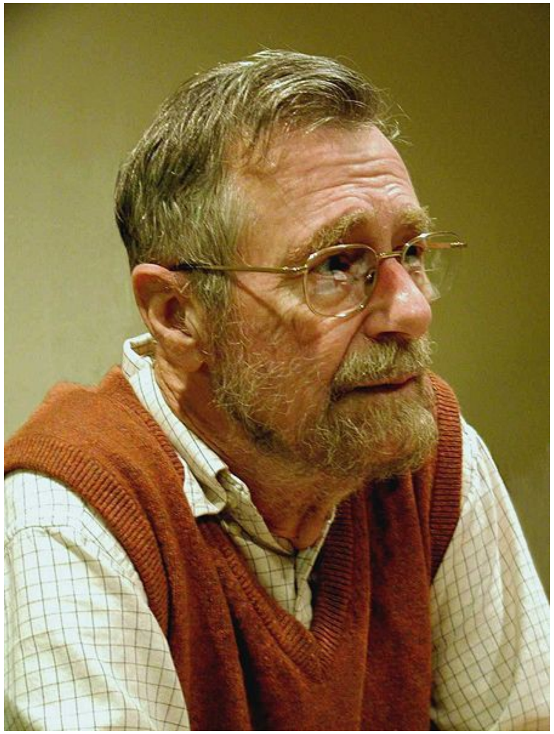
- L'algoritmo di Bellman e Ford determina i cammini di costo minimo anche in presenza di archi con peso negativo
  - Però non devono esistere cicli di peso negativo
  - Il controllo seguente, da fare al termine dell'algoritmo di Bellman e Ford, determina se esistono cicli negativi

```
// eventuale controllo per cicli negativi
for each (u,v) in E do
    if ( D[u] + w(u,v) < D[v] ) then
        error "Il grafo contiene cicli negativi"
    endif
endfor
```

- Nel caso in cui tutti i pesi siano non negativi, esiste un algoritmo più efficiente

Algoritmo di Dijkstra
single-source shortest path

- Determina i cammini di costo minimo da singola sorgente nel caso in cui tutti gli archi abbiano costo ≥ 0

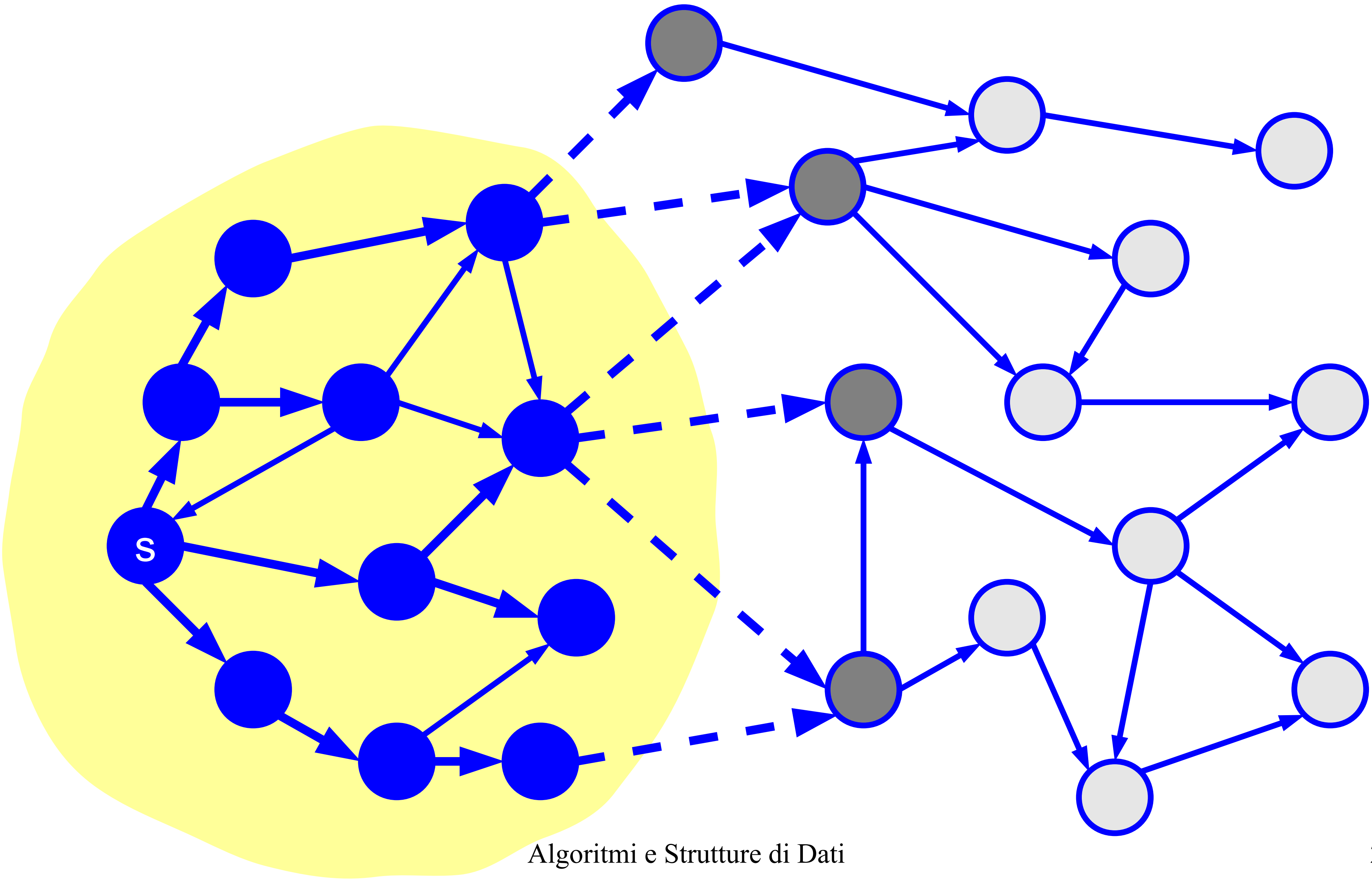


Edsger W. Dijkstra, (1930—2002)
http://en.wikipedia.org/wiki/Edsger_W._Dijkstra

Lemma (Dijkstra)

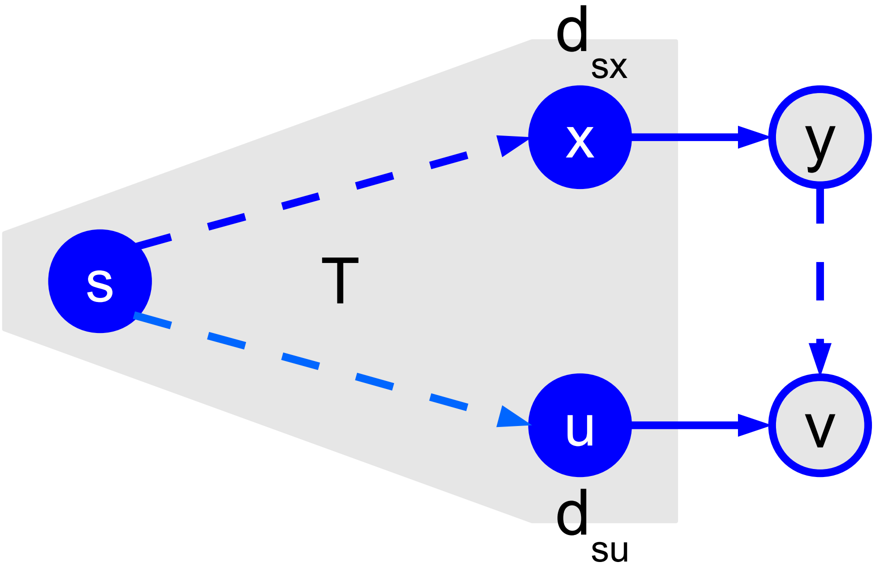
- Sia $G = (V, E)$ un grafo orientato con funzione costo w
  - I costi degli archi devono essere ≥ 0.
- Sia T una parte dell'albero dei cammini di costo minimo radicato in s
  - T rappresenta porzioni di cammini di costo minimo che partono da s
- Allora l'arco (u, v) con $u \in V(T)$ e $v \notin V(T)$ che minimizza la quantità $d_{su} + w(u, v)$ appartiene ad un cammino minimo da s a v

Lemma (Dijkstra)



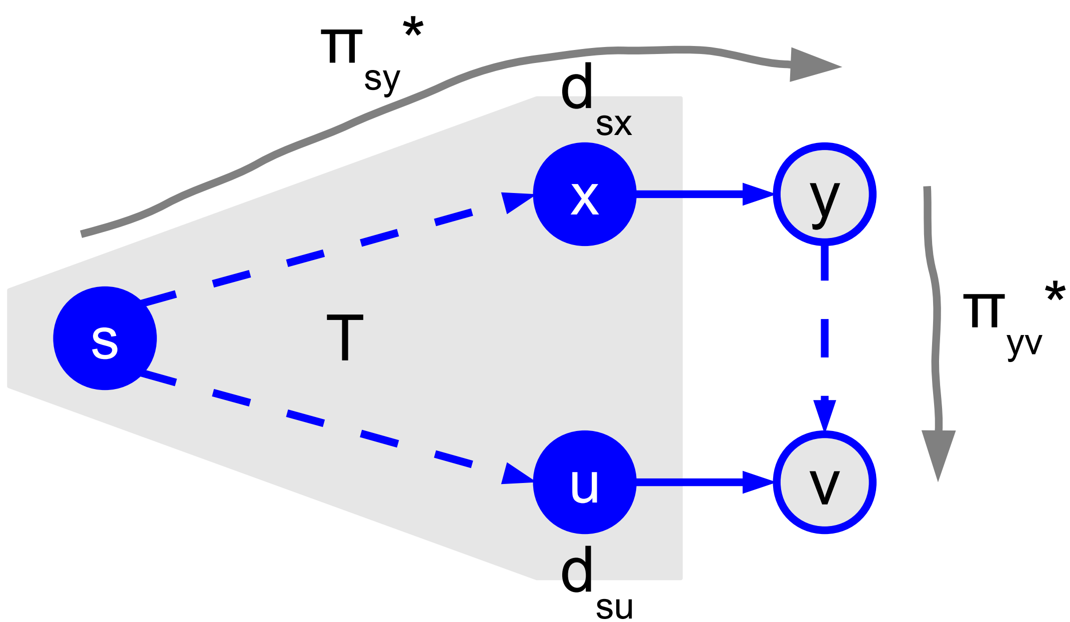
Dimostrazione

- Supponiamo per assurdo che (u,v) *non appartenga* ad un cammino di costo minimo tra s e v
  - quindi $d_{su} + w(u,v) > d_{sv}$ 1
- Quindi deve esistere $\pi_{sv}{}^*$ che porta da s in v senza passare per (u,v) con costo inferiore a $d_{su} + w(u,v)$



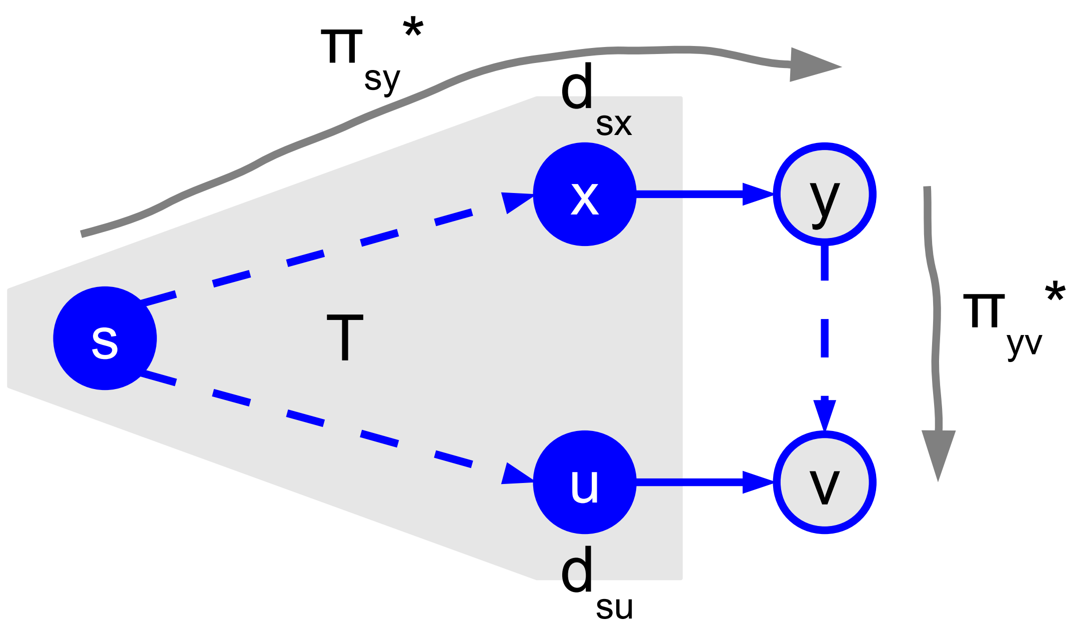
Dimostrazione

- Il cammino π_{sv}* si scompone in π_{sy}* e π_{yv}*, con y primo nodo fuori *T* attraversato dal cammino minimo
- Quindi $d_{sv} = d_{sx} + w(x,y) + d_{yv}$ 2



Dimostrazione

- Per ipotesi (lemma di Dijkstra), l'arco (u,v) è quello che, tra tutti gli archi che collegano un vertice in *T* con uno non ancora in *T*, minimizza la somma $d_{su} + w(u,v)$
- In particolare: $d_{su} + w(u,v) \leq d_{sx} + w(x,y)$ 3



Riassumiamo

- Da (1) abbiamo $d_{su} + w(u,v) > d_{sv}$ 1
- Da (2) abbiamo $d_{sv} = d_{sx} + w(x,y) + d_{yv}$ 2
- Da (3) abbiamo $d_{su} + w(u,v) \leq d_{sx} + w(x,y)$ 3
- Combinando (1) (2) e (3) otteniamo

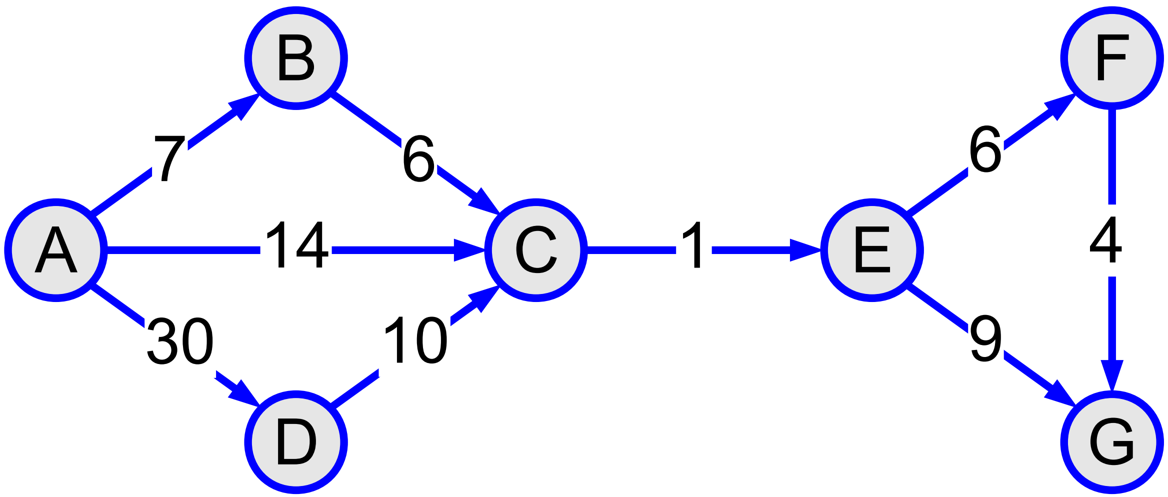
$$
\begin{aligned}
d_{su} + w(u,v) \quad &> \quad d_{sx} + w(x,y) + d_{yv} && \text{da (1) e (2)} \\
&\geq \quad d_{sx} + w(x,y) && \text{da pesi non negativi} \\
&\geq \quad d_{su} + w(u,v) && \text{da (3)}
\end{aligned}
$$

Assurdo!

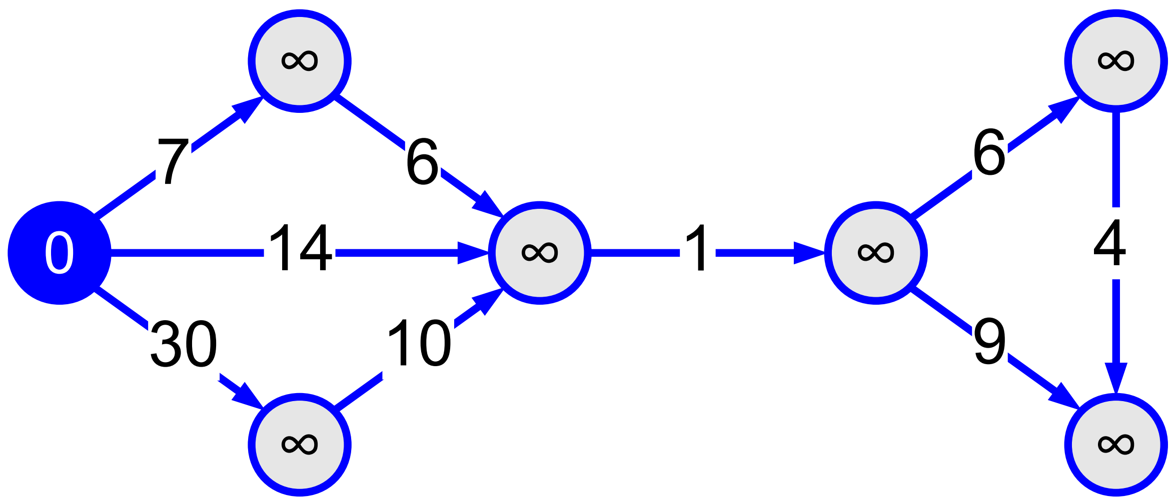
Algoritmo di Dijkstra generico

```
double[1..n] DijkstraGenerico(Grafo G=(V,E,w), int s)
    int n ← G.numNodi();
    int pred[1..n], u, v;
    double D[1..n];
    for v ← 1 to n do
        D[v] ← +∞;
        pred[v] ← -1;
    endfor
    D[s] ← 0;
    while (non ho visitato tutti i nodi raggiungibili da s) do
        Trova l'arco (u,v) incidente su T con D[u] + w(u,v) minimo
        D[v] ← D[u] + w(u,v);
        pred[v] ← u;
    endfor
    return D;
```

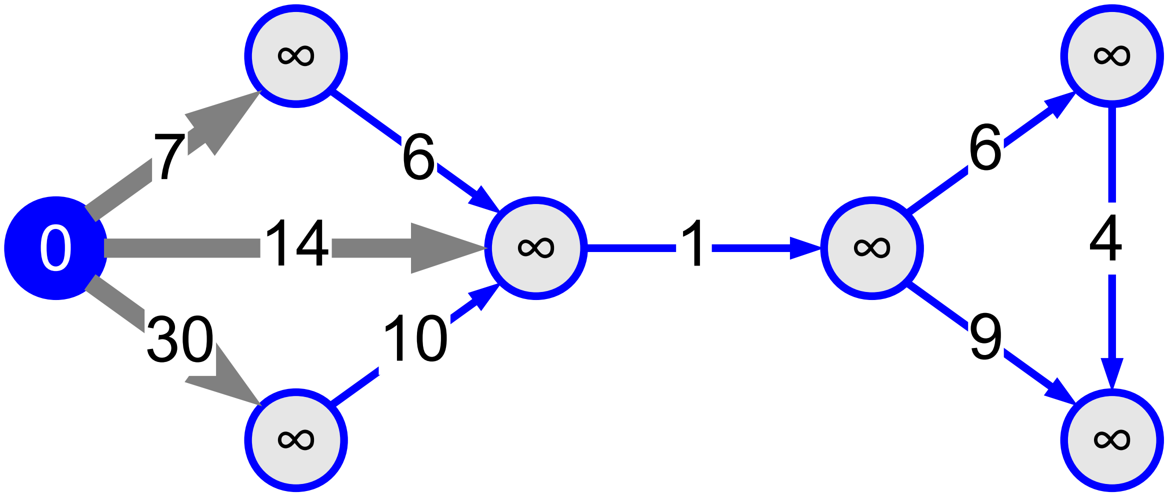
Esempio



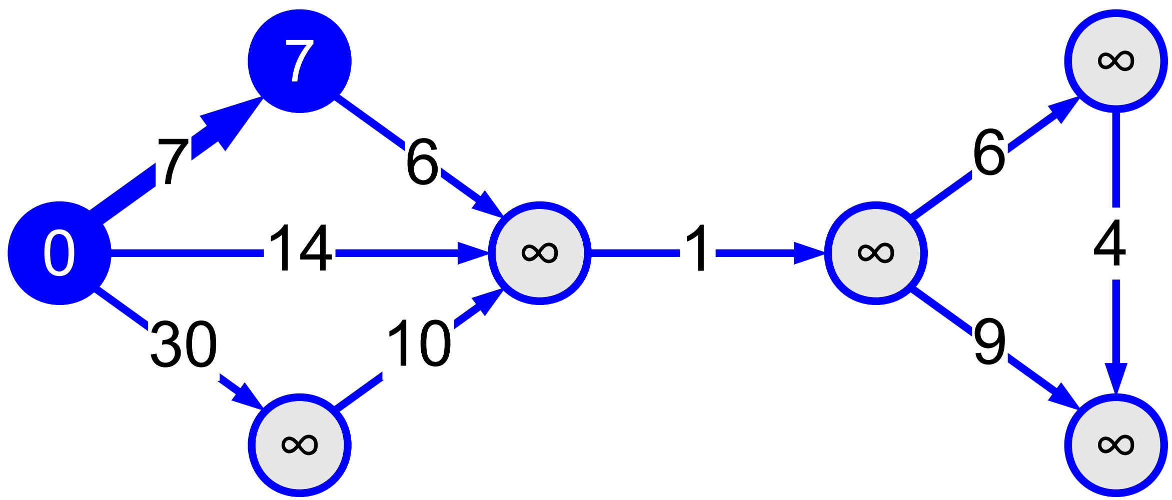
Esempio



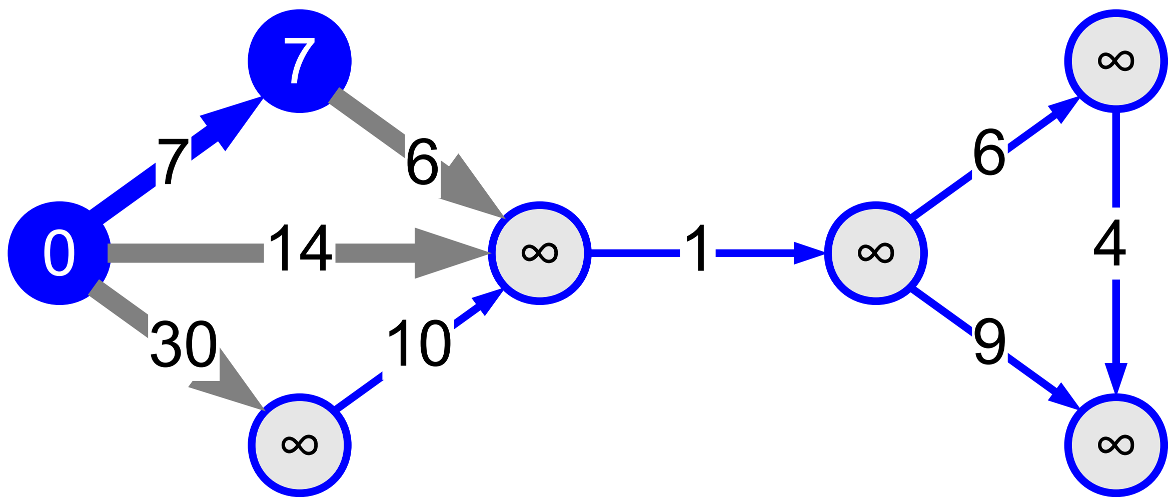
Esempio



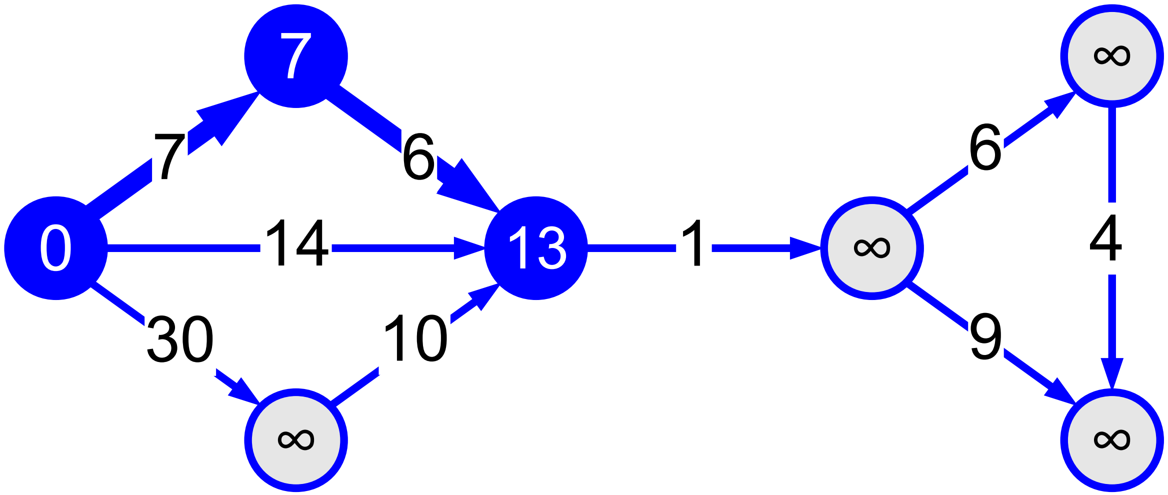
Esempio



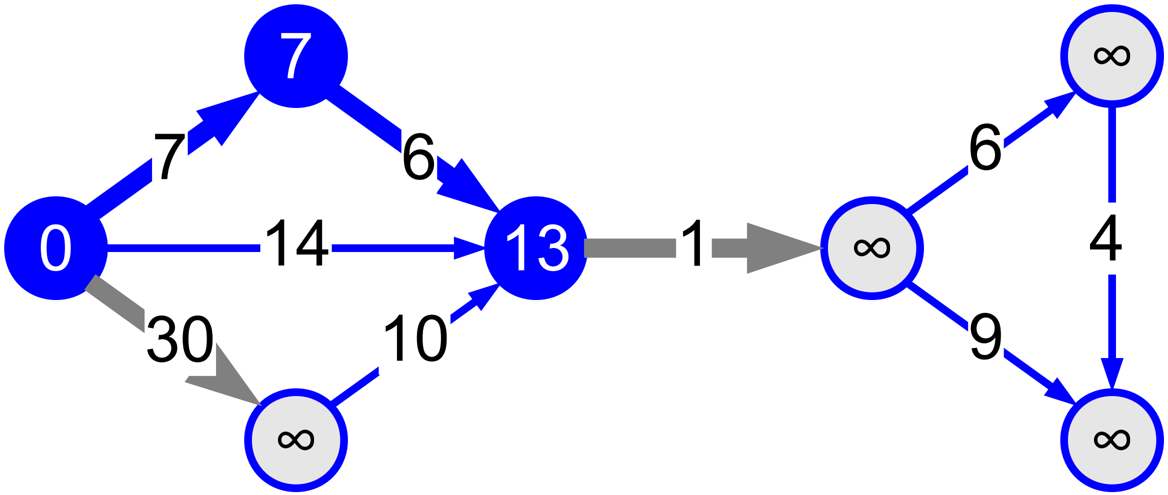
Esempio



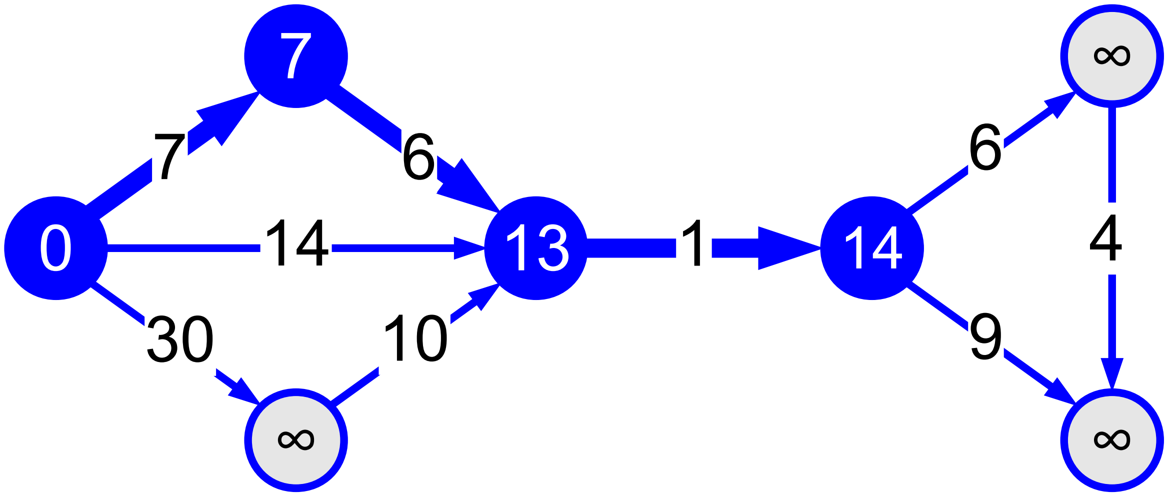
Esempio



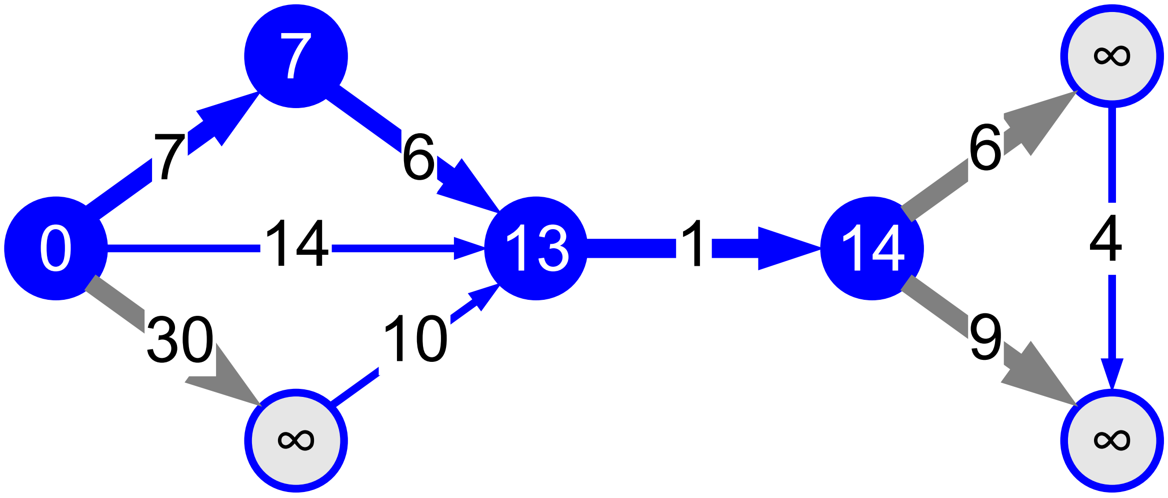
Esempio



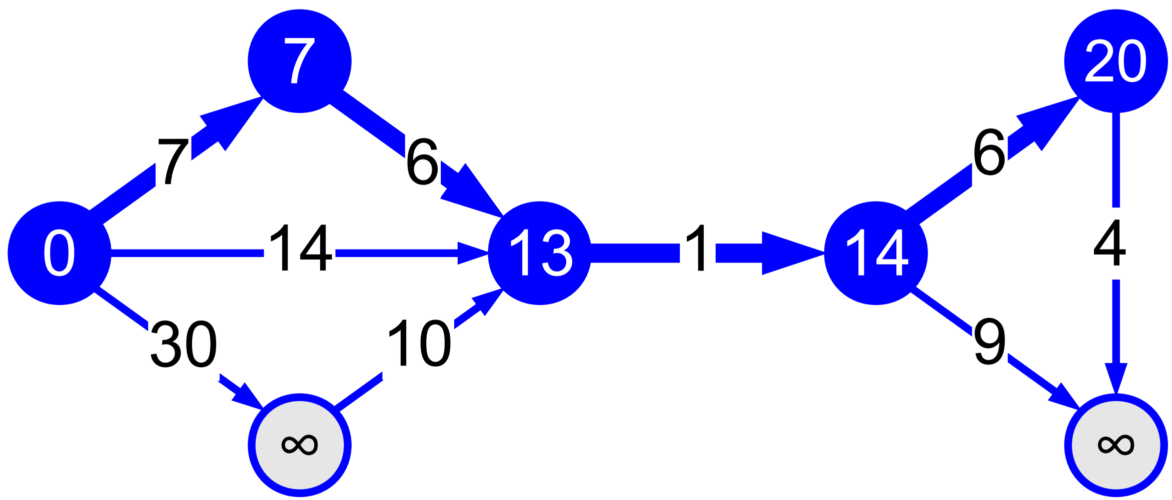
Esempio



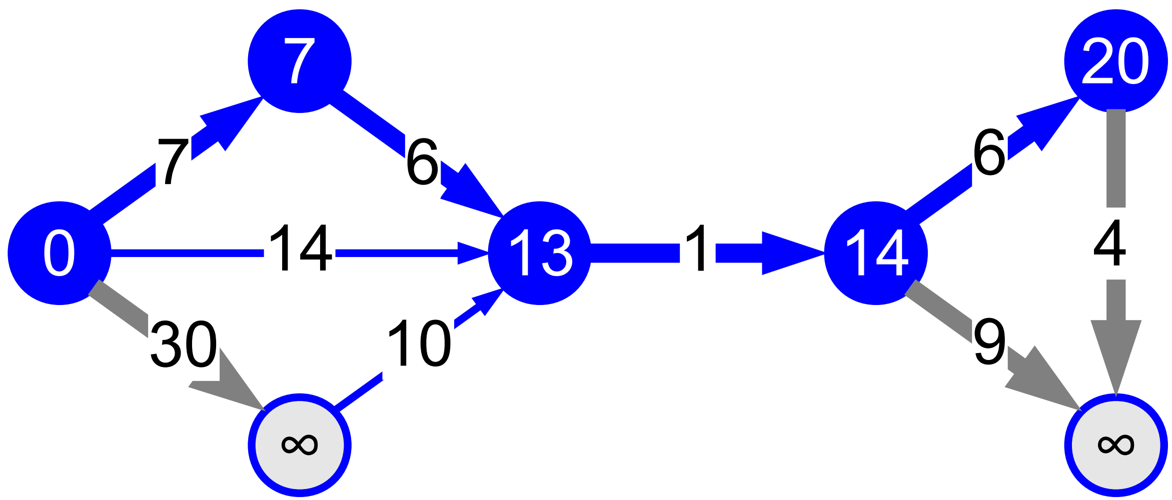
Esempio



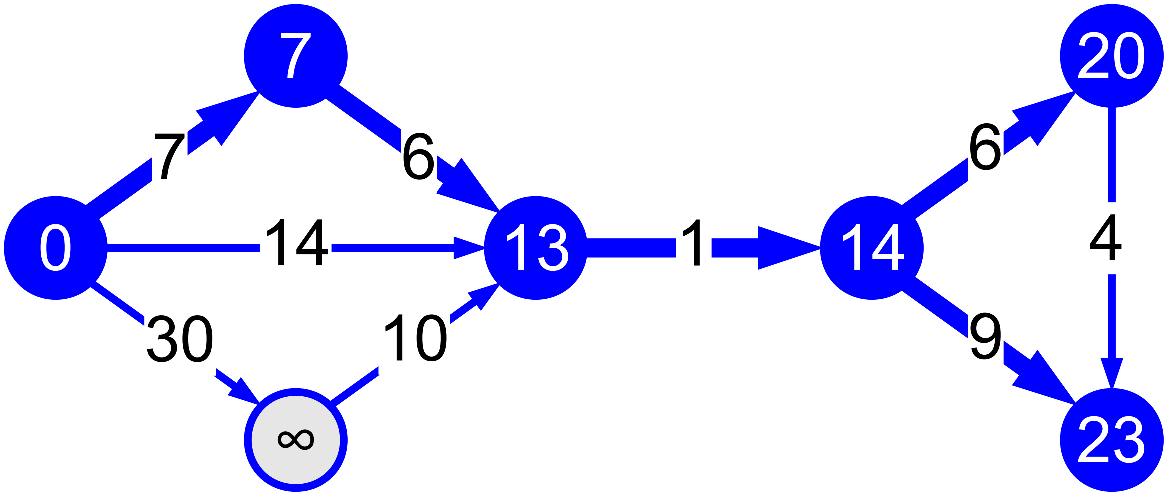
Esempio



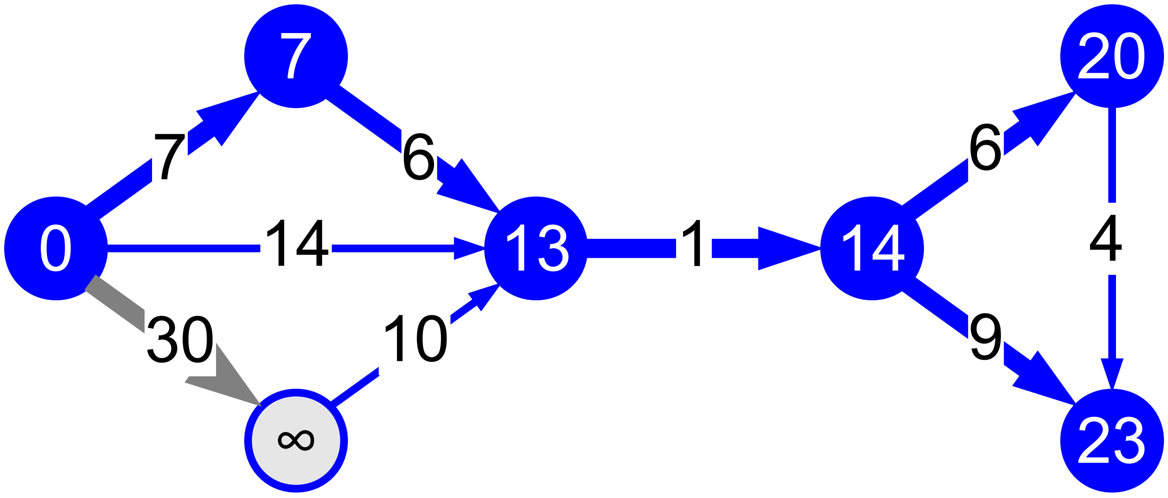
Esempio



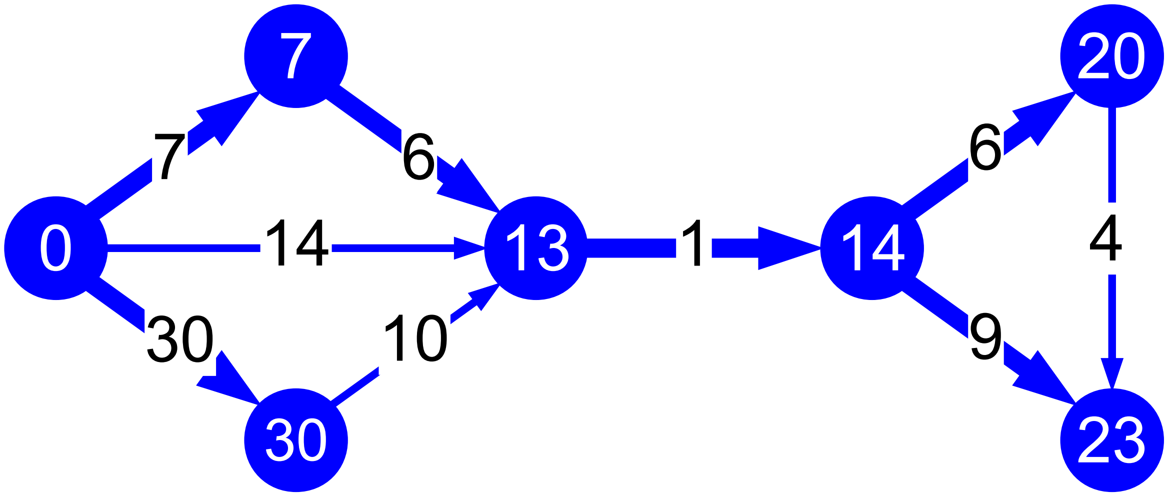
Esempio



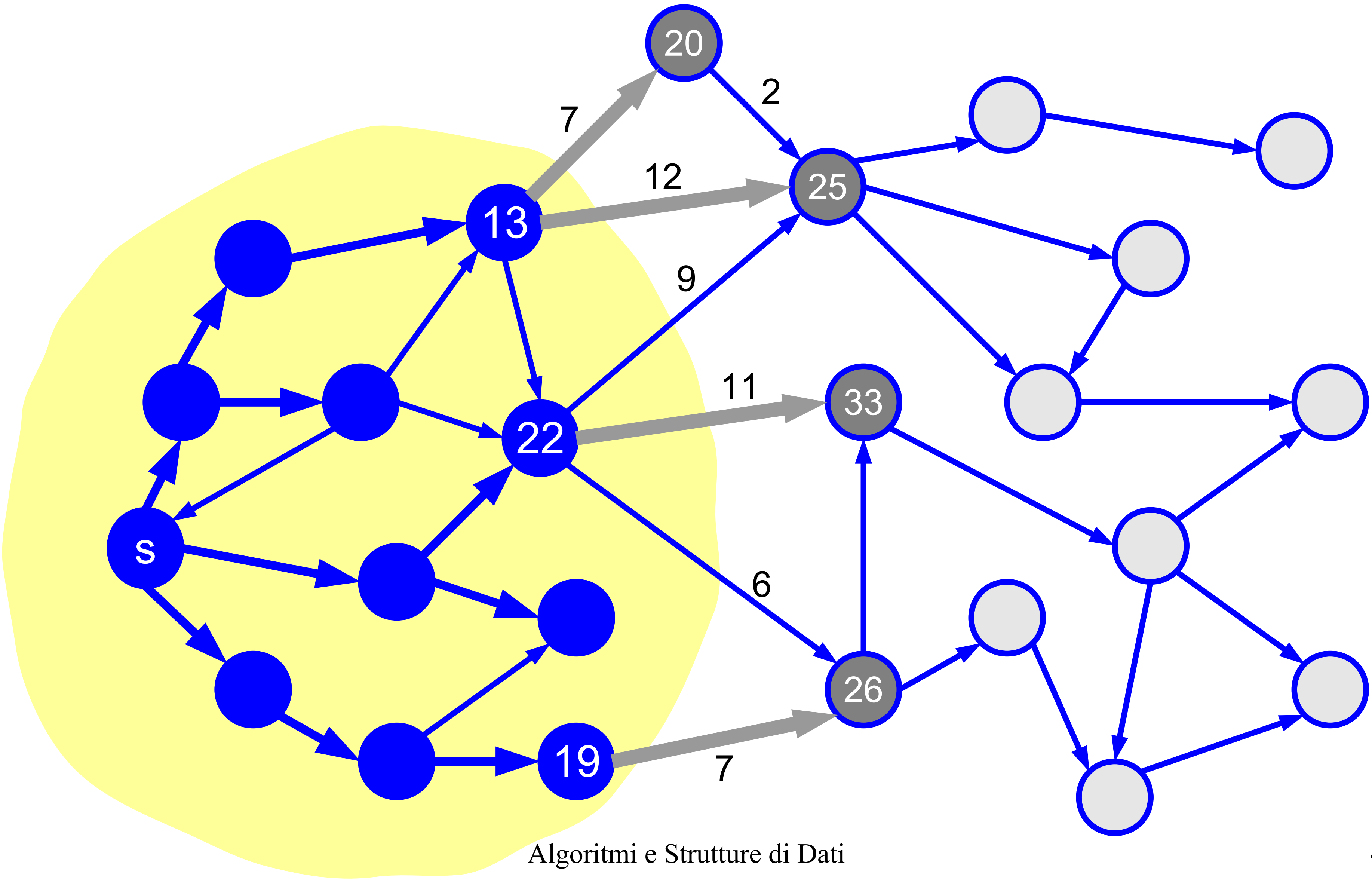
Esempio



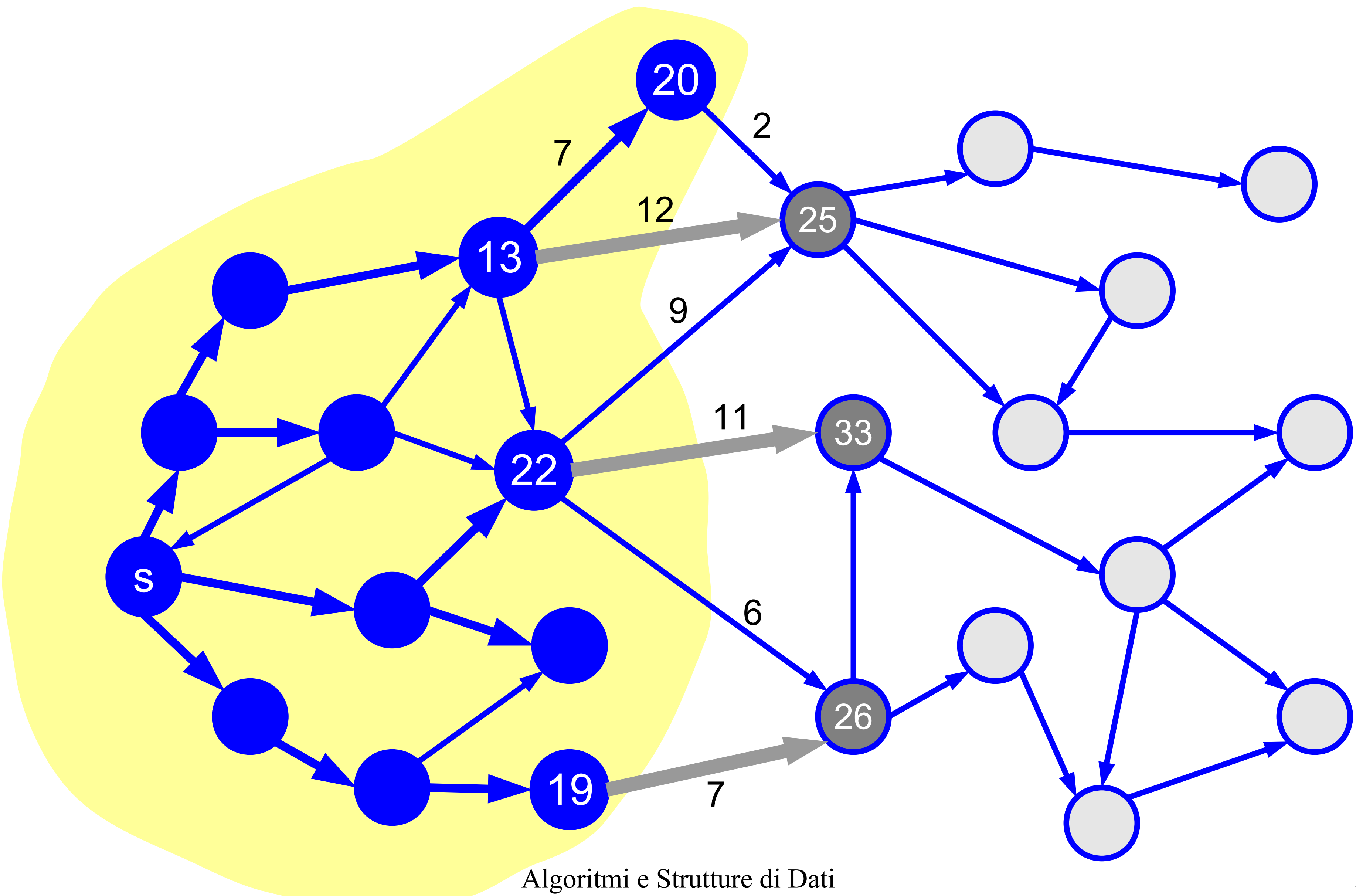
Esempio



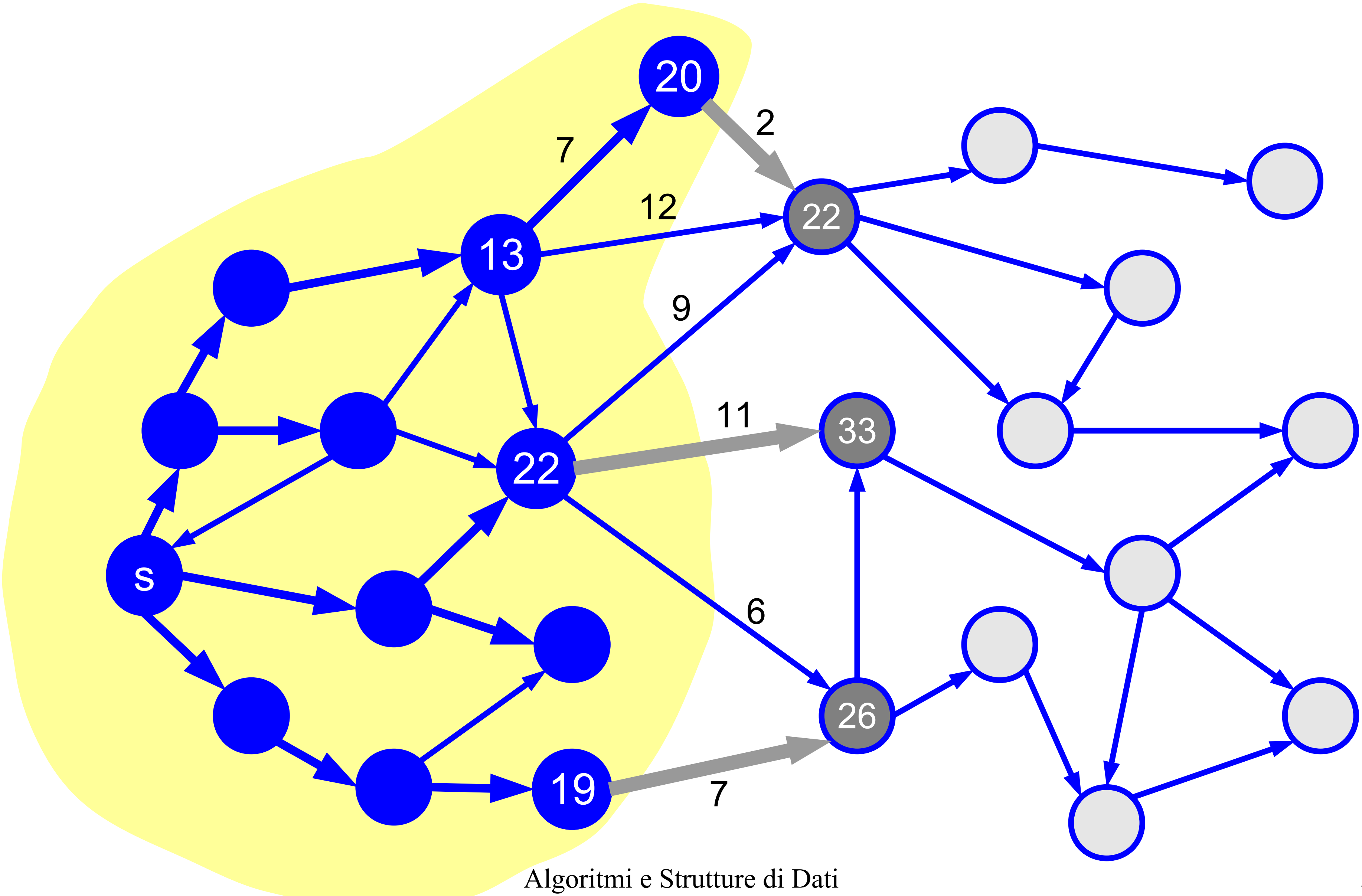
Implementazione



Implementazione



Implementazione



```
double[1..n] Dijkstra(Grafo G=(V,E,w), int s)
    int n ← G.numNodi();
    int pred[1..n], v, u;
    double D[1..n];
    for v ← 1 to n do
        D[v] ← +∞;
        pred[v] ← -1;
    endfor
    D[s] ← 0;
    CodaPriorita<int, double> Q; Q.insert(s, D[s]);
    while (not Q.isEmpty()) do
        u ← Q.find(); Q.deleteMin();
        for each v adiacente a u do
            if (D[v] == +∞) then
                D[v] ← D[u] + w(u,v);
                Q.insert(v, D[v]);
                pred[v] ← u;
            elseif (D[u] + w(u,v) < D[v]) then
                Q.decreaseKey(v, D[v] - D[u] - w(u,v));
                D[v] ← D[u] + w(u,v);
                pred[v] ← u;
            endif
        endfor
    endwhile
    return D;
```

Trova e rimuovi il nodo con distanza minima

Somiglia all'algoritmo di Prim (MST), ma usa una priorita' diversa

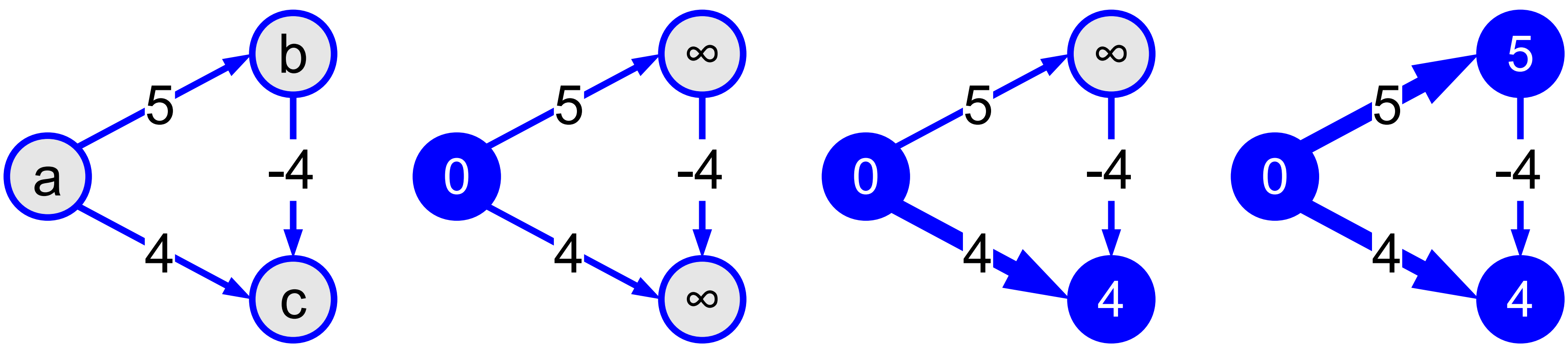
Rendi D[u]+w(u,v) la nuova distanza di v da s

Analisi dell'algoritmo di Dijkstra

- L'inizializzazione ha costo $O(n)$
- Le operazioni find() e deleteMin() hanno costo $O(\log n)$ e sono eseguite al più n volte
  - Una volta che un nodo è stato estratto dalla coda di priorità non verrà più reinserito
- Le operazioni insert() e decreaseKey() hanno costo $O(\log n)$ e sono eseguite al più m volte
  - Una volta per ogni arco
- Totale: $O((n+m) \log n) = O(m \log n)$ se tutti i nodi sono raggiungibili dalla sorgente

Osservazione

- Perché l'algoritmo di Dijkstra funzioni correttamente è essenziale che i pesi degli archi siano tutti ≥ 0
- Esempio di funzionamento errato

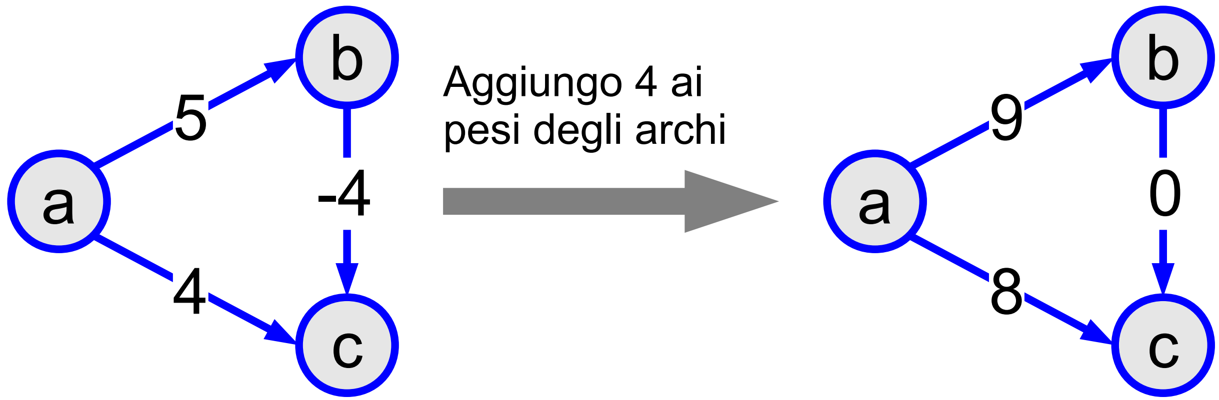


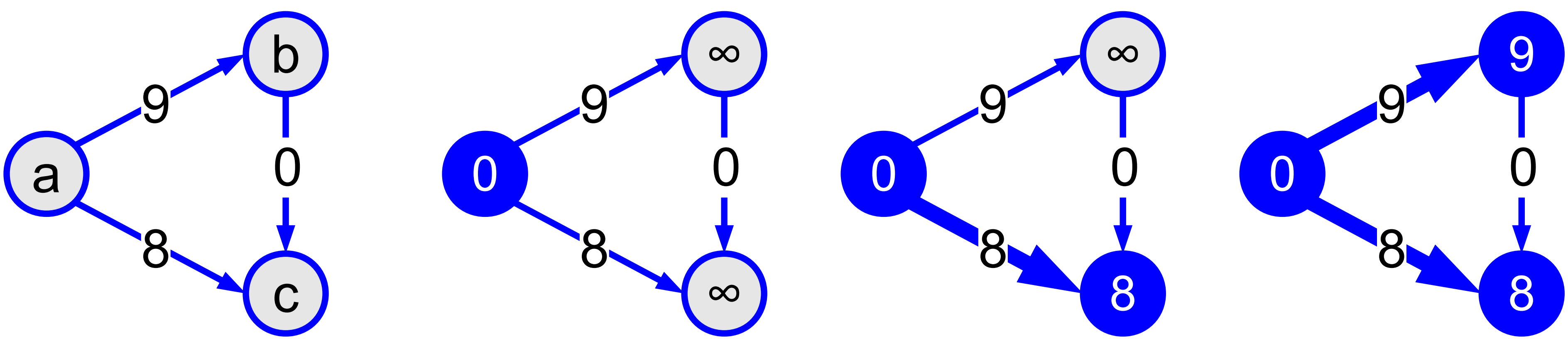
  - Il cammino minimo da a→c non è (a,c) ma (a,b,c) che ha costo 1

Domanda

- Sia $G = (V, E)$ un grafo orientato con archi pesati, anche con pesi negativi; supponiamo che in G non esistano cicli negativi. Supponiamo di incrementare i pesi di tutti gli archi di una costante C in modo che tutti i pesi risultanti siano non negativi.
- L'algoritmo di Dijkstra applicato ai nuovi pesi restituisce l'albero dei cammini minimi anche per i pesi originali?

Risposta: NO


a
b
c
5
-4
4
Aggiungo 4 ai
pesi degli archi
a
b
c
9
0
8


a
b
c
9
0
8
0
∞
∞
9
0
8
0
∞
8
9
0
8
0
9
8
9
0
8

Algoritmo di Floyd e Warshall
all-pair shortest paths

- Si può applicare a grafi orientati con costi arbitrari (anche negativi), purché non ci siano cicli negativi
  - Basato sulla programmazione dinamica
- Sia $V = \{1, 2, \ldots n\}$
- Sia D_{xy}^{k} la distanza minima dal nodo x al nodo y, nell'ipotesi in cui gli eventuali nodi intermedi possano appartenere esclusivamente all'insieme $\{1, \ldots k\}$
- La soluzione al nostro problema è D_{xy}^{n} per ogni coppia di nodi x e y

Inizializzazione

- D_{xy}^0 è la distanza minima tra *x* e *y* nell'ipotesi di non poter passare per alcun nodo intermedio
- Posso calcolare D_{xy}^0 come

Valido sotto assunzione che non esistono cappi con peso negativo

$$D_{xy}^{0} = \begin{cases} 0 & \text{se} \quad x = y \\ w(x, y) & \text{se} \quad (x, y) \in E \\ \infty & \text{se} \quad (x, y) \notin E \end{cases}$$

Caso generale

- Per andare da *x* a *y* usando solo nodi intermedi in {1, … *k*} ho due possibilità
  - Non passo mai per il nodo *k*. La distanza in tal caso è D_{xy}^{k-1}
  - Passo per il nodo *k*. Per la proprietà di sottostruttura ottima, la distanza in tal caso è $D_{x\,k}^{k-1} + D_{k\,y}^{k-1}$
- Quindi

$$D_{xy}^{k} = \min\left\{ D_{xy}^{k-1},\, D_{x\,k}^{k-1} + D_{k\,y}^{k-1} \right\}$$

Algoritmo di Floyd e Warshall

```
double[1..n,1..n] FloydWarshall( G=(V,E,w) )
    int n ← G.numNodi();
    double D[1..n, 1..n, 0..n]; int x, y, k;
    for x ← 1 to n do
        for y ← 1 to n do
            if (x == y) then D[x,y,0] ← 0;
            elseif ((x,y) ∈ E) then D[x,y,0] ← w(x,y);
            else D[x,y,0] ← +∞;
            endif
        endfor
    endfor
    for k ← 1 to n do
        for x ← 1 to n do
            for y ← 1 to n do
                D[x,y,k] ← D[x,y,k-1];
                if (D[x,k,k-1] + D[k,y,k-1] < D[x,y,k] ) then
                    D[x,y,k] ← D[x,k,k-1] + D[k,y,k-1];
                endif
            endfor
        endfor
    endfor
    // eventuale controllo per cicli negativi (vedi seguito)
    return D[1..n, 1..n, n];
```

$$D_{xy}^{k} = \min\left\{ D_{xy}^{k-1}, D_{xk}^{k-1} + D_{ky}^{k-1} \right\}$$

- Costo: tempo $O(n^3)$, spazio $O(n^3)$

Ottimizzazione

```
D[x,y,k] ← D[x,y,k-1];
if (D[x,k,k-1] + D[k,y,k-1] < D[x,y,k] ) then
    D[x,y,k] ← D[x,k,k-1] + D[k,y,k-1];
```

- Al ciclo k-esimo usiamo $D[x,y,k-1]$, $D[x,k,k-1]$ e $D[k,y,k-1]$:
  - le iniziali k-2 matrici non servono!
- Una volta calcolato il nuovo $D[x,y,k]$, il valore $D[x,y,k-1]$ viene utilizzato per calcolare altri $D[x',y',k]$ solo se
  - $x'=x$ e $k=y$ oppure $k=x$ e $y'=y$

  ma in questi casi $D[x,y,k]$ è uguale a $D[x,y,k-1]$ (in quanto $D[x,k,k]=D[x,k,k-1]$ e $D[k,y,k]=D[k,y,k-1]$):
  - una volta calcolato $D[x,y,k]$ il vecchio $D[x,y,k-1]$ non serve più!
- Quindi Floyd e Warshall può funzionare anche usando una sola matrice bidimensionale $D[x,y]$ di n x n elementi:
  - al ciclo k-esimo: $D[x,y] = max \{ D[x,y] , D[x,k]+D[k,y] \}$

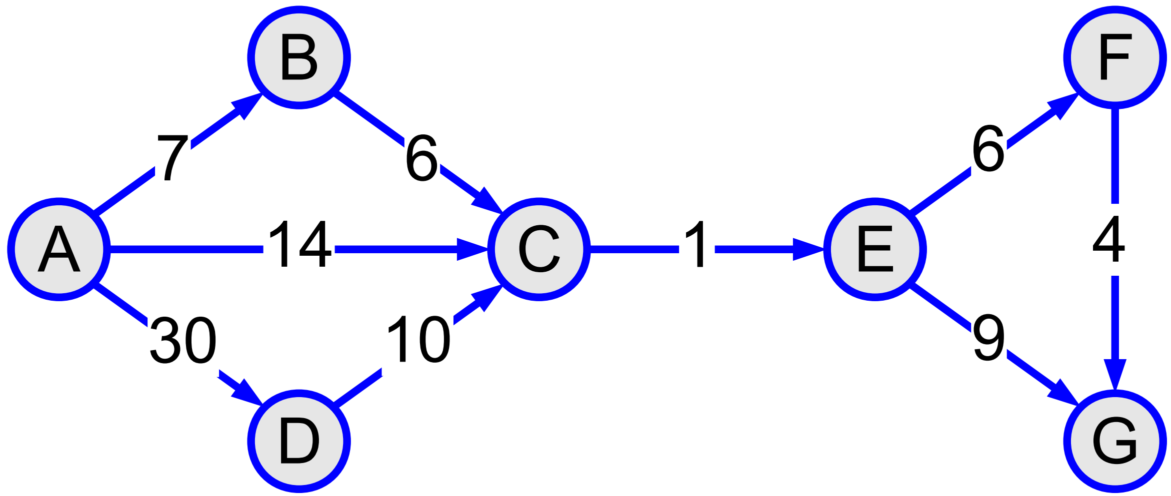
```
double[1..n,1..n] FloydWarshall2( G=(V,E,w) )
    int n ← G.numNodi();
    double D[1..n, 1..n];
    int x, y, k, next[1..n, 1..n];
    for x ← 1 to n do
        for y ← 1 to n do
            if (x == y) then D[x,y] ← 0;
            elseif ((x,y) ∈ E) then D[x,y] ← w(x,y);
            else D[x,y] ← +∞;
            endif
        endfor
    endfor
    for k ← 1 to n do
        for x ← 1 to n do
            for y ← 1 to n do
                if (D[x,k] + D[k,y] < D[x,y]) then
                    D[x,y] ← D[x,k] + D[k,y];
                endif
            endfor
        endfor
    endfor
    return D;
```

Individuare cicli negativi

- Anche l'algoritmo di Floyd e Warshall può essere utilizzato per verificare la presenza di cicli negativi
  - Al termine dell'algoritmo, se $D[x, x, n] < 0$ per qualche x, allora il nodo x fa parte di un ciclo negativo

```
// eventuale controllo per cicli negativi
for x ← 1 to n do
    if ( D[x,x,n] < 0 ) then
        error "Il grafo contiene cicli negativi"
    endif
endfor
```

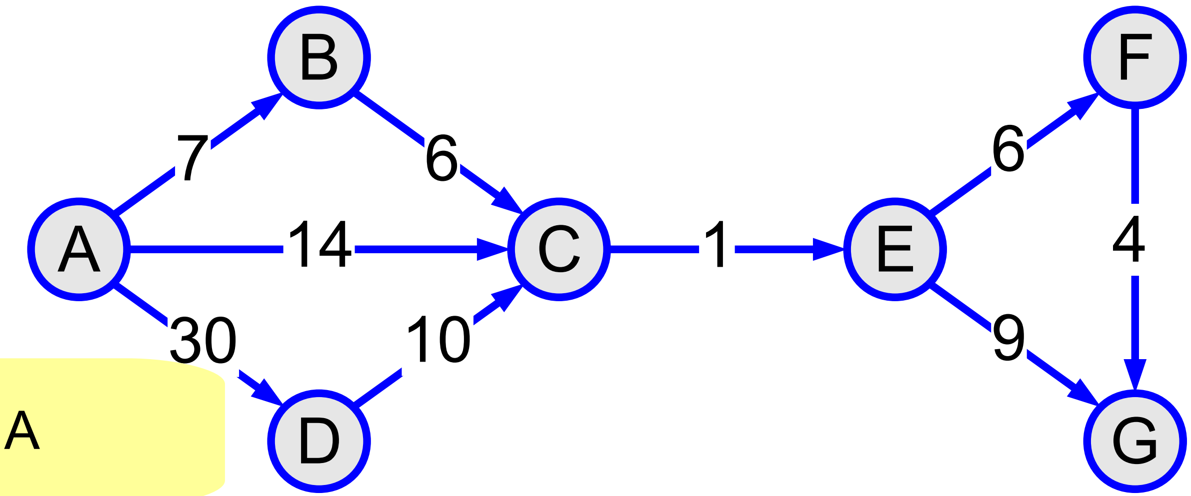
Esempio



D =

	A	B	C	D	E	F	G
A	0	7	14	30	Inf	Inf	Inf
B	Inf	0	6	Inf	Inf	Inf	Inf
C	Inf	Inf	0	Inf	1	Inf	Inf
D	Inf	Inf	10	0	Inf	Inf	Inf
E	Inf	Inf	Inf	Inf	0	6	9
F	Inf	Inf	Inf	Inf	Inf	0	4
G	Inf	Inf	Inf	Inf	Inf	Inf	0

Esempio

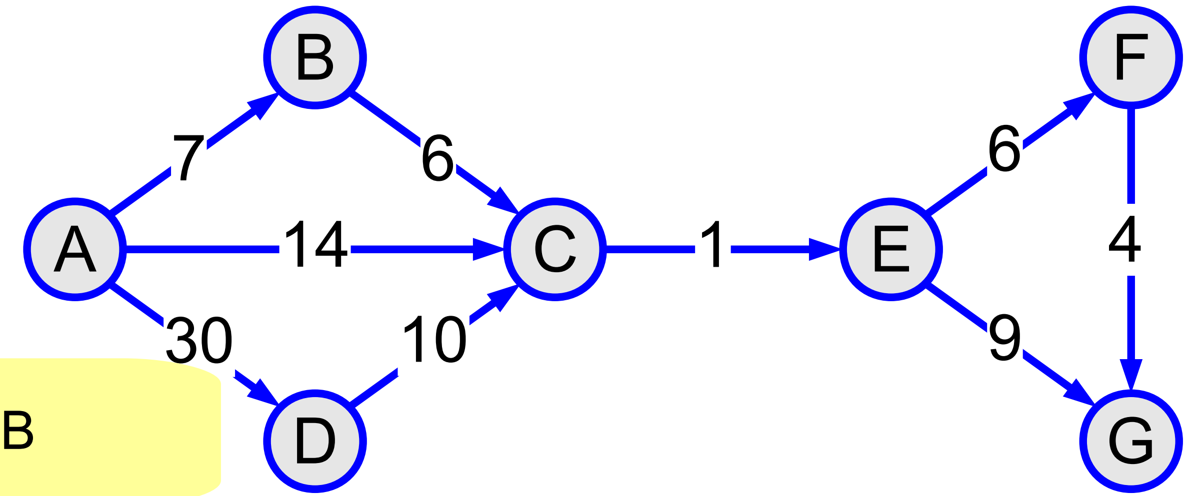


passo in A

D =

	A	B	C	D	E	F	G
A	0	7	14	30	Inf	Inf	Inf
B	Inf	0	6	Inf	Inf	Inf	Inf
C	Inf	Inf	0	Inf	1	Inf	Inf
D	Inf	Inf	10	0	Inf	Inf	Inf
E	Inf	Inf	Inf	Inf	0	6	9
F	Inf	Inf	Inf	Inf	Inf	0	4
G	Inf	Inf	Inf	Inf	Inf	Inf	0

Esempio

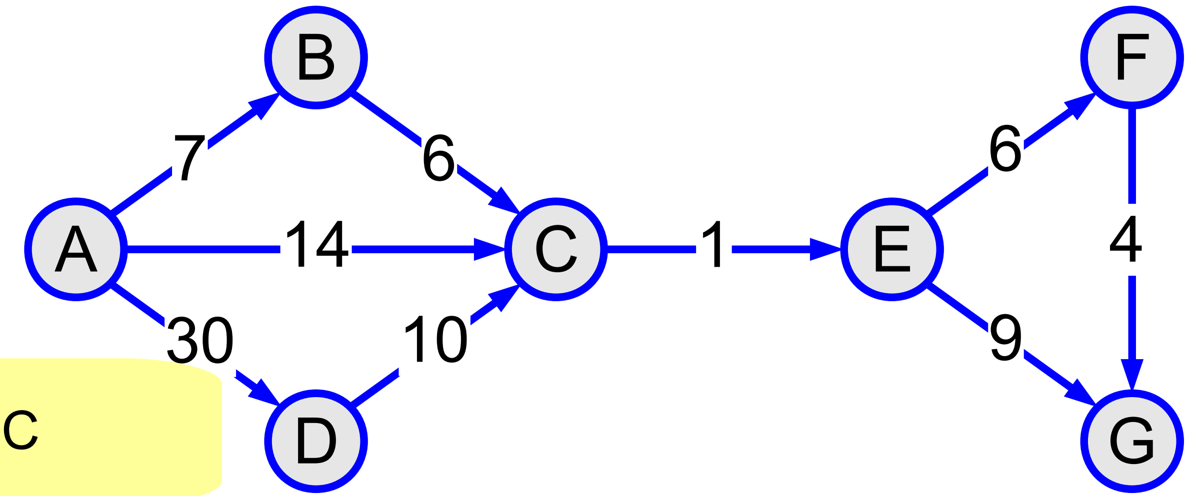


passo in B

D =

	A	B	C	D	E	F	G
A	0	7	13	30	Inf	Inf	Inf
B	Inf	0	6	Inf	Inf	Inf	Inf
C	Inf	Inf	0	Inf	1	Inf	Inf
D	Inf	Inf	10	0	Inf	Inf	Inf
E	Inf	Inf	Inf	Inf	0	6	9
F	Inf	Inf	Inf	Inf	Inf	0	4
G	Inf	Inf	Inf	Inf	Inf	Inf	0

Esempio

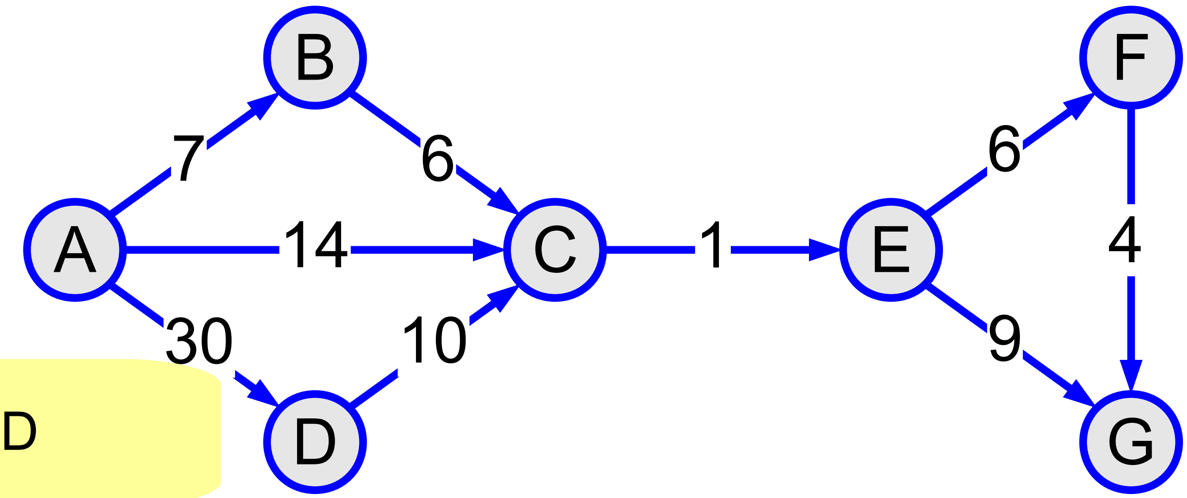


passo in C

D =

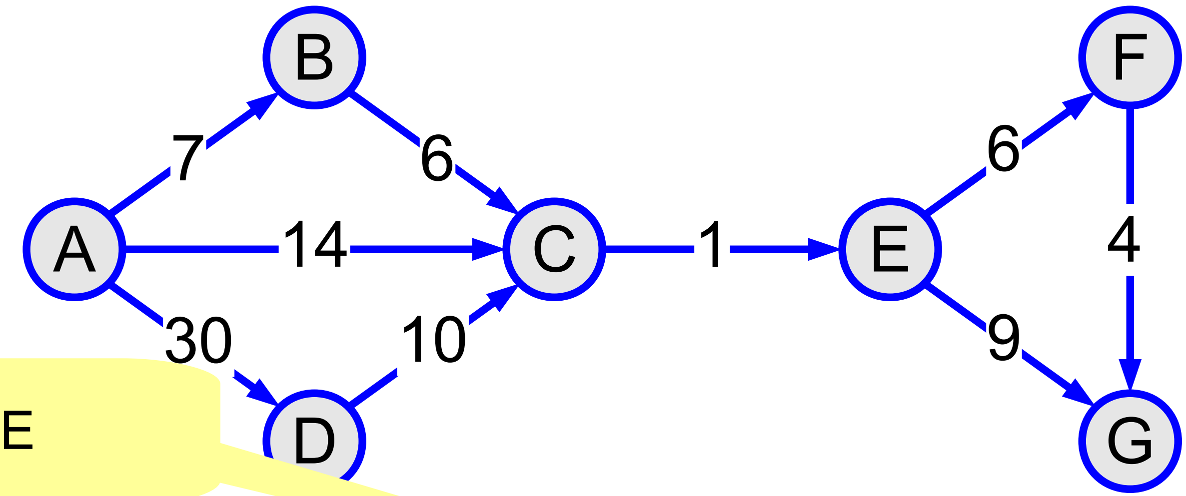
	A	B	C	D	E	F	G
A	0	7	13	30	14	Inf	Inf
B	Inf	0	6	Inf	7	Inf	Inf
C	Inf	Inf	0	Inf	1	Inf	Inf
D	Inf	Inf	10	0	11	Inf	Inf
E	Inf	Inf	Inf	Inf	0	6	9
F	Inf	Inf	Inf	Inf	Inf	0	4
G	Inf	Inf	Inf	Inf	Inf	Inf	0

Esempio




```
D =
         A      B      C      D      E      F      G
A        0      7     13     30     14    Inf    Inf
B      Inf      0      6    Inf      7    Inf    Inf
C      Inf    Inf      0    Inf      1    Inf    Inf
D      Inf    Inf     10      0     11    Inf    Inf
E      Inf    Inf    Inf    Inf      0      6      9
F      Inf    Inf    Inf    Inf    Inf      0      4
G      Inf    Inf    Inf    Inf    Inf    Inf      0
```

Esempio

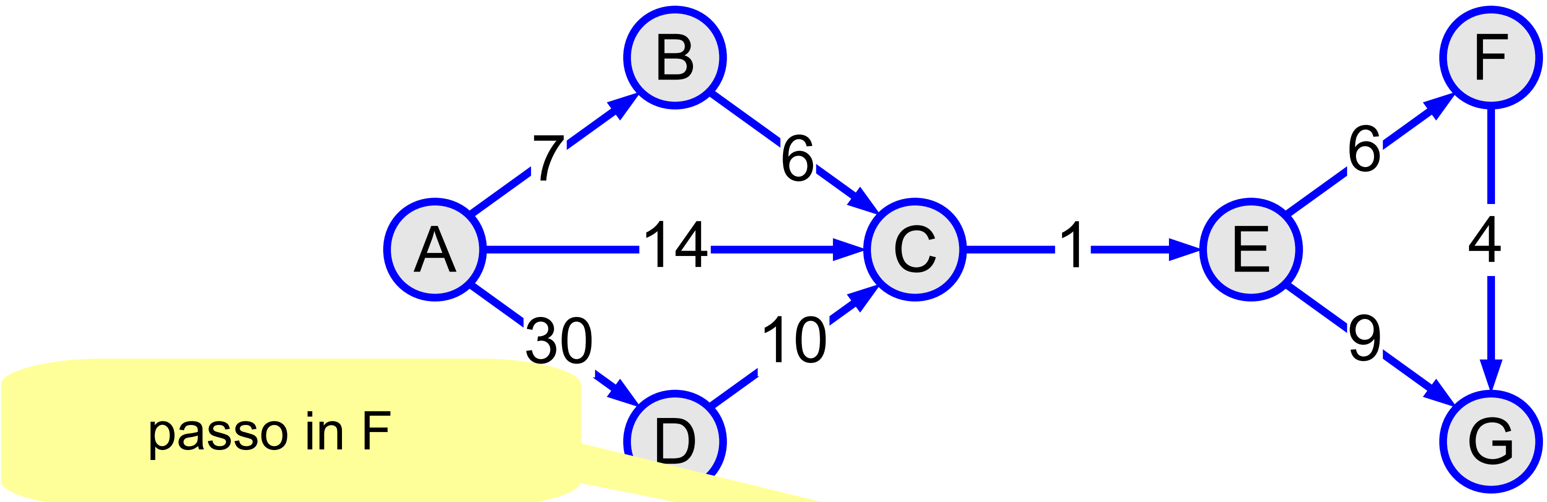


passo in E

D =

	A	B	C	D	E	F	G
A	0	7	13	30	14	20	23
B	Inf	0	6	Inf	7	13	16
C	Inf	Inf	0	Inf	1	7	10
D	Inf	Inf	10	0	11	17	20
E	Inf	Inf	Inf	Inf	0	6	9
F	Inf	Inf	Inf	Inf	Inf	0	4
G	Inf	Inf	Inf	Inf	Inf	Inf	0

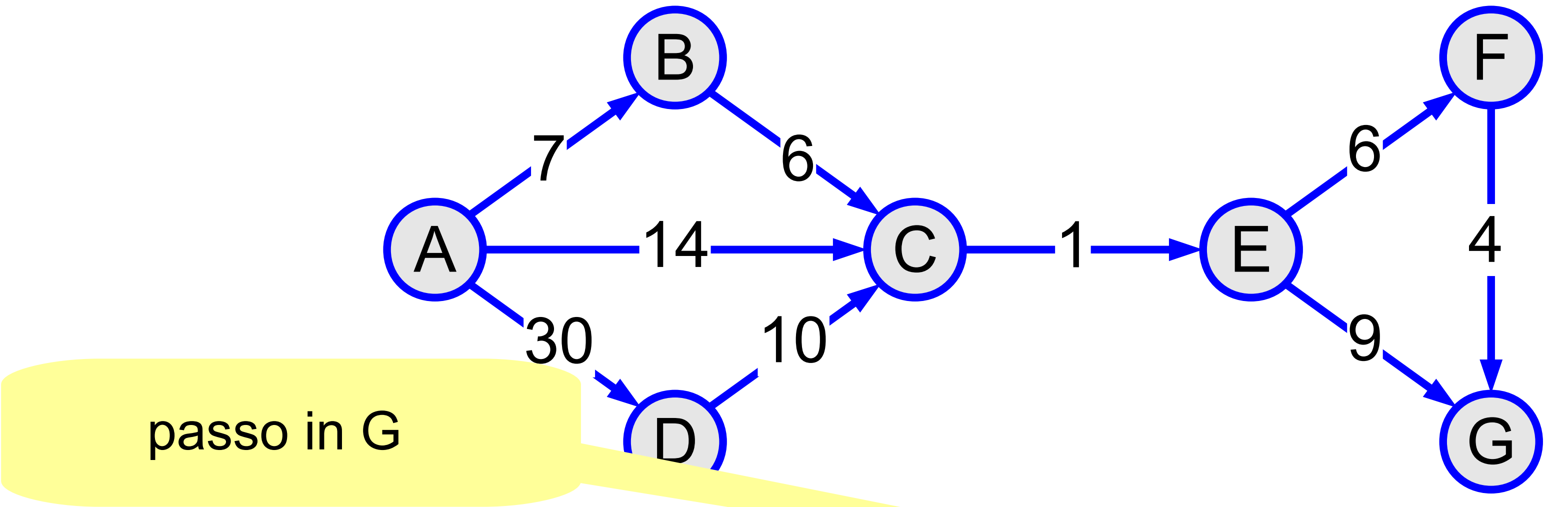
Esempio



D =

	A	B	C	D	E	F	G
A	0	7	13	30	14	20	23
B	Inf	0	6	Inf	7	13	16
C	Inf	Inf	0	Inf	1	7	10
D	Inf	Inf	10	0	11	17	20
E	Inf	Inf	Inf	Inf	0	6	9
F	Inf	Inf	Inf	Inf	Inf	0	4
G	Inf	Inf	Inf	Inf	Inf	Inf	0

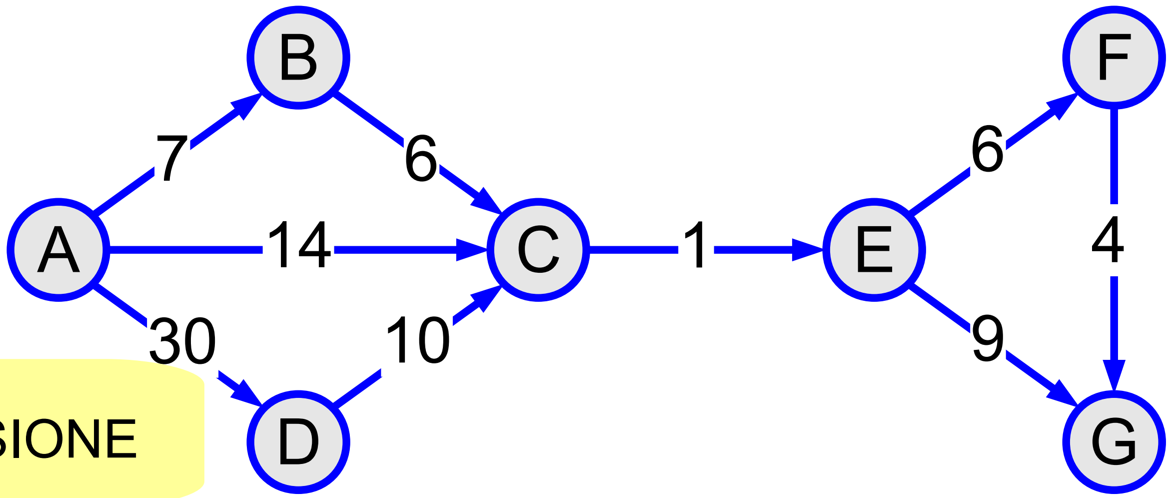
Esempio



D =

	A	B	C	D	E	F	G
A	0	7	13	30	14	20	23
B	Inf	0	6	Inf	7	13	16
C	Inf	Inf	0	Inf	1	7	10
D	Inf	Inf	10	0	11	17	20
E	Inf	Inf	Inf	Inf	0	6	9
F	Inf	Inf	Inf	Inf	Inf	0	4
G	Inf	Inf	Inf	Inf	Inf	Inf	0

Esempio



CONCLUSIONE

D =

	A	B	C	D	E	F	G
A	0	7	13	30	14	20	23
B	Inf	0	6	Inf	7	13	16
C	Inf	Inf	0	Inf	1	7	10
D	Inf	Inf	10	0	11	17	20
E	Inf	Inf	Inf	Inf	0	6	9
F	Inf	Inf	Inf	Inf	Inf	0	4
G	Inf	Inf	Inf	Inf	Inf	Inf	0

Ricostruzione dei cammini

- Per ricostruire i cammini di costo minimo possiamo usare una matrice dei successori *next*[*x*,*y*] di *n* x *n* elementi
  - *next*[*x*,*y*] è l'indice del *secondo* nodo attraversato dal cammino di costo minimo che va da *x* a *y* (il primo nodo di tale cammino è *x*, l'ultimo è *y*)

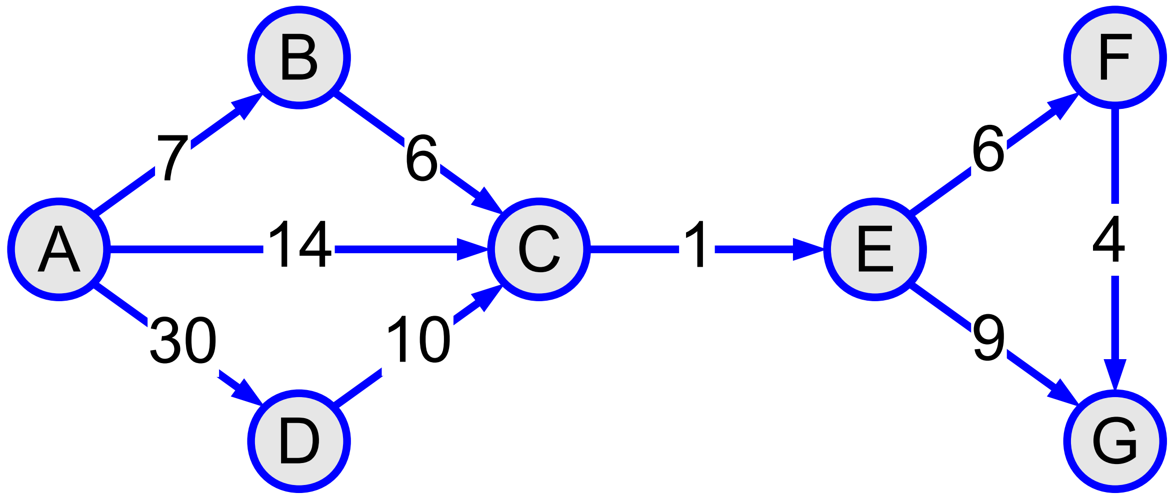
```
double[1..n,1..n] FloydWarshall2( G=(V,E,w) )
    int n ← G.numNodi();
    double D[1..n, 1..n];
    int x, y, k, next[1..n, 1..n];
    for x ← 1 to n do
        for y ← 1 to n do
            if (x == y) then
                D[x,y] ← 0;
                next[x,y] ← -1;
            elseif ((x,y) ∈ E) then
                D[x,y] ← w(x,y);
                next[x,y] ← y;
            else
                D[x,y] ← +∞;
                next[x,y] ← -1;
            endif
        endfor
    endfor
    for k ← 1 to n do
        for x ← 1 to n do
            for y ← 1 to n do
                if (D[x,k] + D[k,y] < D[x,y]) then
                    D[x,y] ← D[x,k] + D[k,y];
                    next[x,y] ← next[x,k];
                endif
            endfor
        endfor
    endfor
    return D;
```

Stampa dei cammini

- Al termine dell'algoritmo di Floyd e Warshall, la procedura seguente stampa i nodi del cammino di costo minimo che va dal nodo *u* al nodo *v* in ordine di attraversamento

```
PrintPath( int u, int v, int next[1..n, 1..n] )
    if ( u != v and next[u,v] < 0 ) then
        errore “u e v non sono connessi”;
    else
        print u;
        while ( u != v ) do
            u ← next[u,v];
            print u;
        endwhile;
    endif
```

Esempio



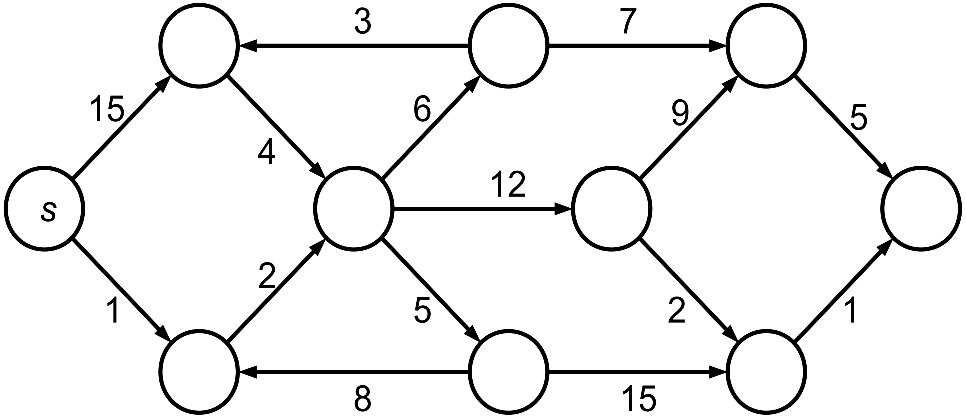
next =

	A	B	C	D	E	F	G
A	-1	B	B	D	B	B	B
B	-1	-1	C	-1	C	C	C
C	-1	-1	-1	-1	E	E	E
D	-1	-1	C	-1	C	C	C
E	-1	-1	-1	-1	-1	F	G
F	-1	-1	-1	-1	-1	-1	G
G	-1	-1	-1	-1	-1	-1	-1

Per rendere la matrice più comprensibile abbiamo usato i nomi dei nodi anziché gli indici

Esercizio 1

- Si consideri il grafo orientato G=(V, E), ai cui archi sono associati costi positivi come illustrato in figura:



  - Applicare “manualmente” l'algoritmo di Dijkstra per calcolare un albero dei cammini di costo minimo partendo dal nodo sorgente *s* (è il nodo più a sinistra in figura).
  - Mostrare il risultato finale dell'algoritmo di Dijkstra, inserendo all'interno di ogni nodo la distanza minima da *s*, ed evidenziando opportunamente (ad esempio, cerchiando il valore del costo) gli archi che fanno parte dell'albero dei cammini di costo minimo.

Esercizio 2

- Si consideri un grafo non orientato G = (V, E), non necessariamente connesso, in cui tutti gli archi abbiano lo stesso peso positivo $\alpha > 0$.
  - Scrivere un algoritmo efficiente che, dato in input il grafo G di cui sopra e due nodi *u* e *v*, calcola la lunghezza del cammino di costo minimo che collega *u* e *v*. Se i nodi non sono connessi, l'algoritmo restituisce $+\infty$.
  - Analizzare il costo asintotico dell'algoritmo proposto.

Esercizio 3

- Una rete stradale è descritta da un grafo orientato pesato $G = (V, E, w)$. Per ogni arco (u, v), la funzione costo $w(u, v)$ indica la quantità di carburante (in litri) che è necessario consumare per percorrere la strada che va da u a v. Tutti i costi sono strettamente positivi. Un veicolo ha il serbatoio in grado di contenere C litri di carburante, inizialmente completamente pieno. Non sono presenti distributori di carburante.
  - Scrivere un algoritmo efficiente che, dati in input il grafo pesato G, la quantità C di carburante inizialmente presente nel serbatoio, e due nodi s e d, restituisce true se e solo se esiste un cammino che consente al veicolo di raggiungere d partendo da s, senza esaurire il carburante durante il tragitto.

Esercizio 4

- Si consideri un grafo orientato pesato, composto dai nodi *{A, B, C, D, E}*, la cui matrice di adiacenza è la seguente (le caselle vuote indicano l'assenza dell'arco corrispondente; l'intestazione di ogni riga indica il nodo sorgente, mentre l'intestazione della colonna indica il nodo destinazione):

	A	***B***	***C***	***D***	***E***
A		2	8		
B			7	9	
C					1
D			-4		
E					

  - Disegnare il grafo corrispondente alla matrice di adiacenza.
  - Determinare la distanza minima di ciascun nodo dal nodo sorgente *A*. Quale degli algoritmi visti a lezione può essere impiegato?